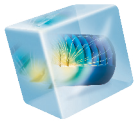


LiveLink™ *for* MATLAB®

User's Guide

VERSION 4.3a

COMSOL

LiveLink™ *for* MATLAB® User's Guide

© 2009–2012 COMSOL

Protected by U.S. Patents 7,519,518; 7,596,474; and 7,623,991. Patents pending.

This Documentation and the Programs described herein are furnished under the COMSOL Software License Agreement (www.comsol.com/sla) and may be used or copied only under the terms of the license agreement.

COMSOL, COMSOL Desktop, COMSOL Multiphysics, and LiveLink are registered trademarks or trademarks of COMSOL AB. MATLAB is a registered trademark of The MathWorks, Inc.. Other product or brand names are trademarks or registered trademarks of their respective holders.

Version: October 2012 COMSOL 4.3a

Contact Information

Visit www.comsol.com/contact for a searchable list of all COMSOL offices and local representatives. From this web page, search the contacts and find a local sales representative, go to other COMSOL websites, request information and pricing, submit technical support queries, subscribe to the monthly eNews email newsletter, and much more.

If you need to contact Technical Support, an online request form is located at www.comsol.com/support/contact.

Other useful links include:

- Technical Support www.comsol.com/support
- Software updates: www.comsol.com/support/updates
- Online community: www.comsol.com/community
- Events, conferences, and training: www.comsol.com/events
- Tutorials: www.comsol.com/products/tutorials
- Knowledge Base: www.comsol.com/support/knowledgebase

Part No. CM020008

Contents

Chapter 1: Introduction

About LiveLink for MATLAB **8**

Help and Documentation **10**
Getting Help 10
Where Do I Access the Documentation and the Model Library? 11
Typographical Conventions 14

Chapter 2: Getting Started

The Client-Server Architecture **18**

Running COMSOL with MATLAB **19**
Starting COMSOL with MATLAB on Windows / Mac OSX / Linux 19
Connecting the COMSOL Server and MATLAB Manually 20
Changing the MATLAB Version 21

Calling a MATLAB Function from the COMSOL Desktop **23**

Chapter 3: Building Models

The Model Object **26**
Important Notes About the Model Object 26
The Model Object Methods 26
The General Utility Functionality 27
Loading and Saving a Model 28
Exchanging Models Between MATLAB and the COMSOL Desktop 29

Working with Geometry **32**
The Geometry Sequence Syntax 32
Displaying the Geometry 33
Working with Geometry Sequences 34
Exchanging Geometries with the COMSOL Desktop 41
Importing and Exporting Geometries and CAD Models from File 42

Retrieving Geometry Information 42
Modeling with a Parameterized Geometry 44
Images and Interpolation Data 46

Working with Meshes **52**
The Meshing Sequence Syntax 52
Displaying the Mesh 53
Mesh Creation Functions 54
Importing External Meshes and Mesh Objects 73
Measuring Mesh Quality 75
Getting Mesh Statistics Information 76
Getting and Setting Mesh Data 77

Modeling Physics **81**
The Physics Interface Syntax 81
The Material Syntax 84
Modifying the Equations 85
Adding Global Equations 86
Defining Model Settings Using External Data File 88

Creating Selections **90**
The Selection Node 90
Coordinate-Based Selections 91
Selection Using Adjacent Geometry 94
Displaying Selections 95

Computing the Solution **97**
The Study Node 97
The Solver Sequence Syntax 99
Run the Solver Sequence 100
Adding a Parametric Sweep 100
Adding a Job Sequence 101
Plot While Solving. 102

Analyzing the Results **104**
The Plot Group Syntax 104
Displaying The Results 105
The Data Set Syntax 108
The Numerical Node Syntax. 109
Exporting Data 110

Chapter 4: Working With Models

Using MATLAB Variables in Model Settings **114**
The Set and SetIndex Methods 114
Using a MATLAB Function to Define Model Properties 115

Extracting Results **117**
Extracting Data From Tables 117
Extracting Data at Node Points 118
Extracting Data at Arbitrary Points 122
Evaluating an Expression at Geometry Vertices 125
Evaluating an Integral 127
Evaluating a Global Expression 129
Evaluating a Global Matrix 131
Evaluating a Maximum of Expression 131
Evaluating an Expression Average 133
Evaluating a Minimum of Expression 135

Running Models in a Loop **137**
The Parametric Sweep Node 137
Running Model in a Loop Using the MATLAB Tools 137

Running Models in Batch Mode **140**
The Batch Node 140
Running A COMSOL M-file In Batch Mode 140
Running A COMSOL M-file In Batch Mode Without Display 141

Working with Matrices **142**
Extracting System Matrices 142
Set System Matrices in the Model 145
Extracting State-Space Matrices 150

Extracting Solution Information and Solution Vector **155**
Obtaining Solution Information 155
Extracting Solution Vector 157

Retrieving Xmesh Information **159**
The Extended Mesh (Xmesh) 159
Extracting Xmesh Information 159

Navigating the Model **162**
Navigating the Model Object Using a GUI 162
Navigating The Model Object At The Command Line 166
Finding Model Expressions 167
Getting Feature Model Properties 168
Getting Model Expressions 168
Getting Selection Information 168

Handling Errors and Warnings **169**
Errors and Warnings 169
Using MATLAB Tools To Handle COMSOL Exceptions 169
Displaying Warnings and Errors in the Model 169

Improving Performance for Large Models **171**
Setting Java Heap Size 171
Disabling Model Feature Update 172
Disabling The Model History 172

Creating a Custom GUI **174**

COMSOL 3.5a Compatibility **175**

Chapter 5: Calling MATLAB Functions

The MATLAB Function Feature Node **178**
Defining a MATLAB Function in the COMSOL Model 178
Setting the Function Directory Path in MATLAB 182
Adding a MATLAB Function with the COMSOL API Java Syntax 183
Function Input/Output Considerations 183
Updating Functions 184
Defining Function Derivatives 184
Using the MATLAB Debugger (Windows Only) 185

Chapter 6: Command Reference

Summary of Commands **188**

Commands Grouped by Function **190**

1

Introduction

This guide introduces you to LiveLink™ *for* MATLAB®, which extends your COMSOL modeling environment with an interface between COMSOL Multiphysics and MATLAB. The *COMSOL Java API Reference Guide* provides additional documentation of the API.

In this chapter:

- About LiveLink for MATLAB
- Help and Documentation

About LiveLink for MATLAB

LiveLink *for* MATLAB connects COMSOL Multiphysics to the MATLAB scripting environment. Using this functionality you can do the following.

SET UP MODELS FROM A SCRIPT

LiveLink *for* MATLAB includes the COMSOL API Java, with all necessary functions and methods to implement models from scratch. For each operation you do in the COMSOL Desktop there is a corresponding command you can enter at the MATLAB prompt. This is a simplified Java-based syntax, which does not require any knowledge of Java. Available methods are listed in the *COMSOL Java API Reference Guide*. The simplest way to learn this programing syntax is to save the model as an M-file directly from the COMSOL Desktop.

You can read more about building a model using the command line in the section Building Models.

USE MATLAB FUNCTIONS IN MODEL SETTINGS

Use LiveLink *for* MATLAB to set model properties with a MATLAB function. For example, define material properties or boundary conditions as a MATLAB routine that is evaluated while the model is solved. This is described in the section Calling MATLAB Functions.

LEVERAGE MATLAB FUNCTIONALITY FOR PROGRAM FLOW

Use the API syntax together with MATLAB functionality to control the flow of your programs. For example, implement nested loops using `for` or `while` commands, or implement conditional model settings with `if` or `switch` statements. You can also handle exceptions using `try` and `catch`. Some of these operations are described in the sections Running Models in a Loop and Handling Errors and Warnings.

ANALYZE RESULTS IN MATLAB

API wrapper functions included with LiveLink *for* MATLAB make it easy to extract data at the command line. Functions are available to access results at node points or arbitrary locations. You can also obtain low level information about the extended mesh, such as finite element mesh coordinates, connection information between the elements, and nodes. Extracted data are available as MATLAB variables, ready to be used with any MATLAB function. See the sections Extracting Results and Retrieving Xmesh Information.

CREATE CUSTOM INTERFACES FOR MODELS

Use the *MATLAB Guide* functionality to create a user-defined graphical interface that is combined with a COMSOL model. Make your models available for others by creating graphical user interfaces tailored to expose settings and parameters of your choice.

Help and Documentation

In this section:

- Getting Help
- Where Do I Access the Documentation and the Model Library?
- Typographical Conventions

Getting Help

COMSOL and LiveLink *for* MATLAB contain several sources of help and information.

THE INTRODUCTION TO LIVELINK FOR MATLAB

To get started with LiveLink *for* MATLAB, it is recommended that you read the *Introduction to LiveLink for* MATLAB. It contains detailed examples about how to get you started with the product.

ONLINE DOCUMENTATION AND OTHER RESOURCES

- Access the on-line documentation with the function `mphdoc`.
- Read this user guide to get detailed information about the different parts of the model object and how these are accessed from MATLAB. The Command Reference chapter describes the function available for use with LiveLink *for* MATLAB.
- The *COMSOL Java API Reference Guide* contains reference documentation that describes the methods in the model object.

M-FILES

Save models as an M-file. Use the COMSOL Desktop to get your first model implemented using the COMSOL Java API.

Set up the model using the graphical user interface, then save the model as an M-file. Next go to the **File** menu and select **Save as M-file**. This generates an M-function that can be run using **COMSOL with MATLAB**.

THE MODEL LIBRARY

Study the LiveLink *for* MATLAB Model Library. LiveLink *for* MATLAB includes a model library with detailed example models. Use the function `mphmodellibrary` at

the command line to get a list of available models. The following are some models that can help get you started.

Model

- Learn how to activate and deactivate domains alternatively during a transient analysis. See the model Domain Activation and Deactivation (model name `domain_activation_llmatlab`).
- Homogenization in a Chemical Reactor (model name `homogenization_llmatlab)` shows how to simulate a periodic homogenization process in a space-dependent chemical reactor model. This homogenization removes concentration gradients in the reactor at a set time interval.
- Convective Heat Transfer with Pseudo-Periodicity (model name `pseudoperiodicity_llmatlab`) simulates convective heat transfer in a channel filled with water. To reduce memory requirements, the model is solved repeatedly on a pseudo-periodic section of the channel. Each solution corresponds to a different section, and before each solution step the temperature at the outlet boundary from the previous solution is mapped to the inlet boundary.
- Temperature Distribution in a Vacuum Flask (model name `vacuum_flask_llmatlab`) shows how to use the MATLAB function callback. This example solves for the temperature distribution inside a vacuum flask with hot coffee.
- Electrical Heating of a Busbar Solved with the LiveLink for SolidWorks and the LiveLink for MATLAB (model name `busbar_llsw_llmatlab`) performs geometry optimization using COMSOL, MATLAB, and SolidWorks.

Where Do I Access the Documentation and the Model Library?

A number of Internet resources provide more information about COMSOL Multiphysics, including licensing and technical information. The electronic

documentation, context help, and the Model Library are all accessed through the COMSOL Desktop.

Important

If you are reading the documentation as a PDF file on your computer, the blue links do not work to open a model or content referenced in a different guide. However, if you are using the online help in COMSOL Multiphysics, these links work to other modules, model examples, and documentation sets.

THE DOCUMENTATION

The *COMSOL Multiphysics User's Guide* and *COMSOL Multiphysics Reference Guide* describe all interfaces and functionality included with the basic COMSOL Multiphysics license. These guides also have instructions about how to use COMSOL Multiphysics and how to access the documentation electronically through the COMSOL Multiphysics help desk.

To locate and search all the documentation, in COMSOL Multiphysics:

- Press F1 or select **Help>Help** ( ) from the main menu for context help.
- Press Ctrl+F1 or select **Help>Documentation** ( ) from the main menu for opening the main documentation window with access to all COMSOL documentation.
- Click the corresponding buttons (or) on the main toolbar.

and then either enter a search term or look under a specific module in the documentation tree.

Tip

If you have added a node to a model you are working on, click the **Help** button ( ) in the node's settings window or press F1 to learn more about it. Under **More results** in the **Help** window, you find a link with a search string for the node's name. Click this link to find all occurrences of the node's name in the documentation, including model documentation and the external COMSOL website. This can help you find more information about the use of the node's functionality as well as model examples where the node has been used.

THE MODEL LIBRARY

Each model comes with documentation that includes a theoretical background and step-by-step instructions to create the model. The models are available in COMSOL

as MPH-files that you can open for further investigation. You can use the step-by-step instructions and the actual models as a template for your own modeling and applications.

SI units are used to describe the relevant properties, parameters, and dimensions in most examples, but other unit systems are available.

To open the Model Library, select **View>Model Library** ( ) from the main menu, and then search by model name or browse under a module folder name. Click to highlight any model of interest, and select **Open Model and PDF** to open both the model and the documentation explaining how to build the model. Alternatively, click the **Help** button ( ) or select **Help>Documentation** in COMSOL to search by name or browse by module.

The model libraries are updated on a regular basis by COMSOL in order to add new models and to improve existing models. Choose **View>Model Library Update** ( ) to update your model library to include the latest versions of the model examples.

If you have any feedback or suggestions for additional models for the library (including those developed by you), feel free to contact us at info@comsol.com.

CONTACTING COMSOL BY EMAIL

For general product information, contact COMSOL at info@comsol.com.

To receive technical support from COMSOL for the COMSOL products, please contact your local COMSOL representative or send your questions to support@comsol.com. An automatic notification and case number is sent to you by email.

COMSOL WEB SITES

Main Corporate web site	www.comsol.com
Worldwide contact information	www.comsol.com/contact
Technical Support main page	www.comsol.com/support
Support Knowledge Base	www.comsol.com/support/knowledgebase
Product updates	www.comsol.com/support/updates
COMSOL User Community	www.comsol.com/community

Typographical Conventions

All COMSOL user's guides use a set of consistent typographical conventions that make it easier to follow the discussion, understand what you can expect to see on the graphical user interface (GUI), and know which data must be entered into various data-entry fields.

In particular, these conventions are used throughout the documentation:

CONVENTION	EXAMPLE
text highlighted in blue	Click text highlighted in blue to go to other information in the PDF. When you are using the online help desk in COMSOL Multiphysics, these links also work to other modules, model examples, and documentation sets.
boldface font	A **boldface** font indicates that the given word(s) appear exactly that way on the COMSOL Desktop (or, for toolbar buttons, in the corresponding tip). For example, the **Model Builder** window ( ) is often referred to and this is the window that contains the model tree. As another example, the instructions might say to click the **Zoom Extents** button ( ), and this means that when you hover over the button with your mouse, the same label displays on the COMSOL Desktop.
italic font	An italic font indicates the introduction of important terminology. Expect to find an explanation in the same paragraph or in the Glossary. The names of other user guides in the COMSOL documentation set also have an italic font.
Forward arrow symbol **>**	The forward arrow symbol **>** is instructing you to select a series of menu items in a specific order. For example, **Options>Preferences** is equivalent to: From the **Options** menu, choose **Preferences**.

CONVENTION	EXAMPLE
`Code` (monospace) font	A `Code` (monospace) font indicates you are to make a keyboard entry in the user interface. You might see an instruction such as "Enter (or type) 1.25 in the **Current density** field." The monospace font also is an indication of programming code or a variable name.
Italic *`Code`* (monospace) font	An italic *`Code`* (monospace) font indicates user inputs and parts of names that can vary or be defined by the user.
Arrow brackets <> following the `Code` (monospace) or *`Code`* (italic) fonts	The arrow brackets included in round brackets after either a monospace Code or an italic Code font means that the content in the string can be freely chosen or entered by the user, such as feature tags. For example, `model.geom(<tag>)` where *<tag>* is the geometry's tag (an identifier of your choice). When the string is predefined by COMSOL, no bracket is used and this indicates that this is a finite set, such as a feature name.

KEY TO THE GRAPHICS

Throughout the documentation, additional icons are used to help navigate the information. These categories are used to draw your attention to the information based on the level of importance, although it is always recommended that you read these text boxes.

ICON	NAME	DESCRIPTION
	Caution	A Caution icon is used to indicate that the user should proceed carefully and consider the next steps. It might mean that an action is required, or if the instructions are not followed, that there will be problems with the model solution.
	Important	An Important icon is used to indicate that the information provided is key to the model building, design, or solution. The information is of higher importance than a note or tip, and the user should endeavor to follow the instructions.
	Note	A Note icon is used to indicate that the information may be of use to the user. It is recommended that the user read the text.
	Tip	A Tip icon is used to provide information, reminders, shortcuts, suggestions of how to improve model design, and other information that may or may not be useful to the user.

ICON	NAME	DESCRIPTION
	See Also	The See Also icon indicates that other useful information is located in the named section. If you are working on line, click the hyperlink to go to the information directly. When the link is outside of the current PDF document, the text indicates this, for example "See The Laminar Flow Interface in the *COMSOL Multiphysics User's Guide*." Note that if you are in COMSOL Multiphysics' online help, the link works.
	Model	The Model icon is used in the documentation as well as in COMSOL Multiphysics from the View>Model Library menu. If you are working online, click the link to go to the PDF version of the step-by-step instructions. In some cases, a model is only available if you have a license for a specific module. These examples occur in the COMSOL Multiphysics User's Guide. The Model Library path describes how to find the actual model in COMSOL Multiphysics, for example If you have the RF Module, see Radar Cross Section: Model Library path **RF_Module/Tutorial_Models/radar_cross_section**
Space Dimension		Another set of icons are also used in the Model Builder—the model space dimension is indicated by 0D , 1D , 1D axial symmetry , 2D , 2D axial symmetry , and 3D icons. These icons are also used in the documentation to clearly list the differences to an interface, feature node, or theory section, which are based on space dimension.

2

Getting Started

In this chapter:

- The Client-Server Architecture
- Running COMSOL with MATLAB
- Calling a MATLAB Function from the COMSOL Desktop

The Client-Server Architecture

LiveLink *for* MATLAB uses the client-server mode to connect COMSOL Multiphysics and MATLAB. When starting COMSOL with MATLAB, two processes are started—a COMSOL server and the MATLAB desktop. The MATLAB process is a client connected to the COMSOL server using a TCP /IP communication protocol.

Note

The COMSOL Desktop is not involved.

The first time you start COMSOL with MATLAB, you are requested to provide login information. This information is stored in the user preferences file and is not requested for later use of COMSOL with MATLAB. The same login information may be used when exchanging the model object between the COMSOL server and the COMSOL Desktop.

The communication between the COMSOL server and MATLAB is established by default using port number 2036. If this port is in use, port number 2037 is used instead, and so on.

See Also

You can manually specify the port number. See COMSOL Server Commands in the *COMSOL Multiphysics Installation and Operations Guide* for more information on the COMSOL server start-up properties.

Important

The links to features described outside of this user guide do not work in the PDF, only from within the online help.

Tip

To locate and search all the documentation for this information, in COMSOL, select **Help>Documentation** from the main menu and either enter a search term or look under a specific module in the documentation tree.

Running COMSOL with MATLAB

The command to run COMSOL with MATLAB automatically connects a COMSOL process with MATLAB. You can also connect the process manually. This section describes how to automatically and manually start COMSOL with MATLAB as well as how to change the MATLAB path in the COMSOL settings.

Note: The *System Requirements* section of the *COMSOL Installation and Operations Guide* lists the versions of MATLAB supported by LiveLink *for* MATLAB.

In this section:

- Starting COMSOL with MATLAB on Windows / Mac OSX / Linux
- Connecting the COMSOL Server and MATLAB Manually
- Changing the MATLAB Version

Starting COMSOL with MATLAB on Windows / Mac OSX / Linux

To run a COMSOL model at the MATLAB prompt, first start COMSOL with MATLAB.

- On Windows use the **COMSOL with MATLAB** shortcut icon that is created on the desktop after the automatic installation. A link is also available in the Windows start menu under **All Programs>COMSOL 43a>COMSOL 4.3a with MATLAB**.
- On Mac OS X, use the **COMSOL with MATLAB** application available in the application folder.
- On Linux enter the command `comsol server matlab` at a terminal window.

See Also

See the *COMSOL Multiphysics Installation and Operations Guide* to get a complete description on how to start COMSOL with MATLAB on the different supported platforms.

Note

The first time **COMSOL with MATLAB** is started, login and password information is requested to establish the client/server connection. The information is then saved in the user preference file and is not requested again.

If you want to reset the login information, add the flag `-login force` to the icon target path on Windows. For Mac OS X and Linux operating systems, enter the command `comsol server matlab -login force` at a system command prompt.

Connecting the COMSOL Server and MATLAB Manually

Manually connecting MATLAB to a COMSOL server can be useful if you need to start a MATLAB standalone and then connect to a COMSOL server, or if you need to connect MATLAB and a COMSOL server running on different computers.

To manually connect MATLAB to a COMSOL server you need first to start MATLAB and a COMSOL server.

Starting a COMSOL Server

- On Windows go to the start menu **All Programs>COMSOL 4.3a> Client Server>COMSOL Multiphysics 4.3a server**.
- On Mac OS X or Linux enter `comsol server` at a terminal window.

Connecting MATLAB to the COMSOL Server

1. In MATLAB, add the path of the COMSOL43a/mli directory.
2. Enter the command below at the MATLAB prompt:

```
mphstart(<portnumber>)
```

Where `<portnumber>` is the port used by the COMSOL server. If the COMSOL server is listening on the default port, `2036`, the port number does not need to be specified.

ADJUSTING THE MATLAB JAVA HEAP SIZE

You may need to modify the MATLAB Java heap size to be able to manipulate the model object and extract data at the MATLAB prompt. See Improving Performance for Large Models.

CONNECTING MATLAB AND THE COMSOL SERVER ON DIFFERENT COMPUTERS

Note

This operation requires the specific license type called Floating Network License (FNL).

To connect MATLAB and a COMSOL server that are running on different computers, specify in the function `mphstart` the IP address of the computer where the COMSOL server is running:

```
mphstart(<ipaddress>, <portnumber>)
```

IMPORTING THE COMSOL CLASS

Once MATLAB and the COMSOL server are manually connected, import the COMSOL class by entering the following command at the MATLAB prompt:

```
import com.comsol.model.*
import com.comsol.model.util.*
```

Disconnecting MATLAB and the COMSOL Server

To disconnect MATLAB and the COMSOL server, run the command below at the MATLAB prompt:

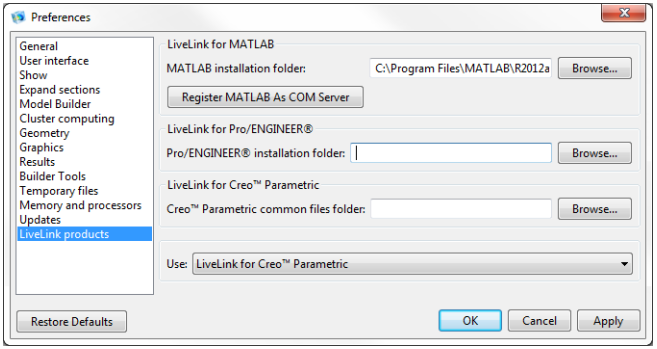
```
ModelUtil.disconnect;
```

Changing the MATLAB Version

The path of the MATLAB version connected to COMSOL is defined during the initial COMSOL installation. The MATLAB root path can be changed using the preferences file:

1. In the COMSOL Desktop, go to the **Options** menu and select **Preferences**.
2. In the **Preferences** window, go to **LiveLink products**.
3. Set the MATLAB root directory path in the **MATLAB installation folder** field.
4. Windows OS users also need to click **Register MATLAB as COM Server** button, otherwise the specified MATLAB version may not start when calling external MATLAB function from the COMSOL model.

5 Click **OK**.



6 To update the preferences file, close the COMSOL Desktop.

Calling a MATLAB Function from the COMSOL Desktop

Use LiveLink *for* MATLAB to call MATLAB functions from within the model when working in the COMSOL Desktop. The procedure is slightly different than implementing a model using a script as you do not need to run COMSOL with MATLAB.

Start COMSOL as a standalone application. The external MATLAB function needs to be defined in the COMSOL model so that a MATLAB process can automatically start when the function needs to be evaluated. The result of the function evaluation in MATLAB is then sent back to the COMSOL environment.

See Also

Calling MATLAB Functions

3

Building Models

This chapter gives an overview of the model object and provides an introduction to building models using the LiveLink interface.

In this chapter:

- The Model Object
- Working with Geometry
- Working with Meshes
- Modeling Physics
- Creating Selections
- Computing the Solution
- Analyzing the Results

The Model Object

While working with the LiveLink interface in MATLAB you work with models through the *model object*. Use *methods* to create, modify, and access your model.

In this section:

- Important Notes About the Model Object
- The Model Object Methods
- The General Utility Functionality
- Loading and Saving a Model
- Exchanging Models Between MATLAB and the COMSOL Desktop

Important Notes About the Model Object

The following information should be considered regarding the model object:

- All algorithms and data structures for the model are integrated in the model object.
- The model object is used by the COMSOL Desktop to represent your model. This means that the model object and the COMSOL Desktop behavior are virtually identical.
- The model object includes methods for setting up and running *sequences of operations* to create geometry, meshes, and for solving your model.

LiveLink *for* MATLAB includes the COMSOL Java API, which is a Java-based programming interface to COMSOL. In addition, the product includes a number of M-file utility functions that wrap API functionality for greater ease of use.

The Model Object Methods

The model object provides a large number of methods. The methods are structured in a tree-like way, very similar to the nodes in the model tree in the *Model Builder* window on the COMSOL Desktop. The top-level methods just return references that

support further methods. At a certain level the methods perform actions, such as adding data to the model object, performing computations, or returning data.

Detailed documentation about model object methods is About General Commands in the *COMSOL Java API Reference Guide*.

The links to features described outside of this user guide do not work in the PDF, only from within the online help.

To locate and search all the documentation for this information, in COMSOL, select **Help>Documentation** from the main menu and either enter a search term or look under a specific module in the documentation tree.

The General Utility Functionality

The model object utility methods are available with the *ModelUtil* object. These methods can be used, for example, to create or remove a new model object, but also to enable the progress bar or list the model object available in the COMSOL server.

MANAGING THE COMSOL MODEL OBJECT

Use the method `ModelUtil.create` to create a new model object in the COMSOL server:

```
model = ModelUtil.create('Model');
```

This command creates a model object `Model` on the COMSOL server and a MATLAB object `model` that is linked to the model object.

It is possible to have several model objects on the COMSOL server, each with a different name. To access each model object requires different MATLAB variables linked to them and each MATLAB variable must have a different name.

Create a MATLAB variable linked to an existing model object with the method `ModelUtil.model`. For example, to create a MATLAB variable `model` that is linked to the existing model object `Model` on the COMSOL server, enter the command:

```
model = ModelUtil.model('Model');
```

To remove a specific model object use the method `ModelUtil.remove`. For example, to remove the model object `Model` from the COMSOL server enter the command:

```
ModelUtil.remove('Model');
```

Alternatively remove all the COMSOL objects stored in the COMSOL server with the command:

```
ModelUtil.clear
```

List the names of the model objects available on the COMSOL server with the command:

```
list = ModelUtil.tags
```

ACTIVATING THE PROGRESS BAR

By default no progress information is displayed while running COMSOL with MATLAB. To manually enable a progress bar and visualize the progress of operations (such as loading a model, creating a mesh, assembling matrices, or computing the solution), enter the command:

```
ModelUtil.showProgress(true);
```

To deactivate the progress bar enter:

```
ModelUtil.showProgress(false);
```

Note

The progress bar is not supported on Mac OS X.

Loading and Saving a Model

LOADING A MODEL AT THE MATLAB PROMPT

To load an existing model saved as an MPH-file use the function `mphload`. For example to load the Busbar model from the Model Library enter:

```
model = mphload('busbar.mph');
```

This creates a model object `Model` on the COMSOL server that is accessible using the MATLAB variable `model`.

If there is already a model object `Model` linked to a MATLAB variable `model`, load the model using a different name with the command:

```
model2 = mphload('busbar.mph','Model2');
```

When using the function `mphload`, the model history is automatically disabled to prevent large history information when running a model in a loop. To turn model history on use the function `mphload`:

```
model = mphload('busbar.mph','-history');
```

The history recording can be useful when using the COMSOL Desktop. All the operations are then stored in the saved model M-file.

SAVING A MODEL OBJECT

Use the function `mphsave` to save the model object linked to the MATLAB object model:

```
mphsave(model,'filename')
```

If the filename specified `'filename'` does not provide a path the file is saved relative to the local MATLAB path. The file extension determines the format to use (`*.mph`, `*.m`, or `*.java`).

Alternatively, use the `save` method:

```
model.save('filename');
```

If `'filename'` does not provide a path, the file is saved relative to the local COMSOL server path.

Any files saved in the MPH format can be loaded by the COMSOL Desktop. In addition, the model can be saved as a Model M-file:

```
model.save('model_name','m');
```

Note

The models are not automatically saved between MATLAB sessions.

Exchanging Models Between MATLAB and the COMSOL Desktop

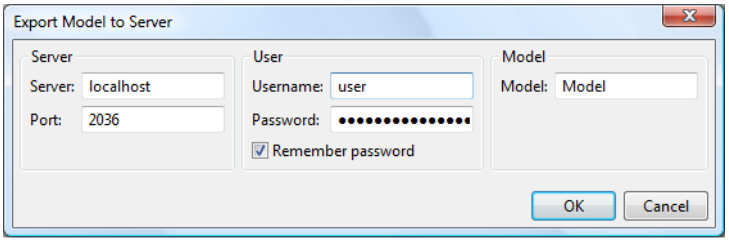
It is possible to alternate between the MATLAB scripting interface and the COMSOL graphical user interface in order to edit or modify the model object. When running

COMSOL with MATLAB, the model object is stored on the COMSOL server. You can directly load the model object in the COMSOL Desktop from the COMSOL server or, conversely, export the model object available in the COMSOL Desktop to the COMSOL server.

EXPORTING FROM THE COMSOL DESKTOP A MODEL TO MATLAB

In the COMSOL Desktop, use the option **Export to Server** from the **File** menu to send the model object to the COMSOL server that is connected with MATLAB. Once the model object is on the COMSOL server, create a link in the MATLAB prompt.

1. Open a model in the COMSOL Desktop.
2. From the **File** menu choose **Client Server>Export Model to Server**.



3. Make sure that the **Server** (the default is `localhost`) and the **Port** (the default is `2036`) fields are set with the correct information. To establish the connection between the COMSOL Desktop and the COMSOL server enter a **Username** and a **Password**; these are defined the first time you are connected to the COMSOL server.
4. Enter the name of the model object to export (the default name is `Model`).
5. Click **OK**.
6. In MATLAB, create a link to the model object on the COMSOL server with the command:

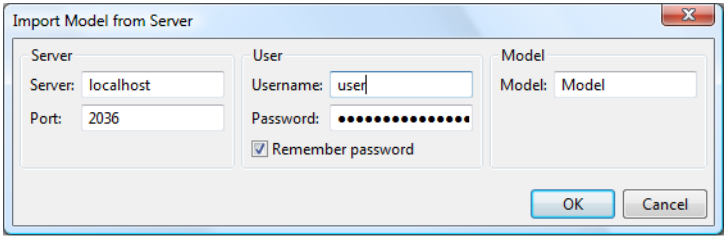
```
model = ModelUtil.model('Model');
```

Note

If the model has been exported to the COMSOL server using a different name, replace `Model` with the correct name of the exported model. Use the command: `ModelName = ModelUtil.tags` to obtain the list of model objects available on the COMSOL server.

IMPORTING A MODEL IN THE COMSOL DESKTOP FROM MATLAB

To import a model from a COMSOL server to the COMSOL Desktop, from the **File** menu choose **Client Server>Import Model from Server**. This dialog box is similar to the **Client Server>Export Model to Server** dialog box.



Important

The COMSOL server may hold several models; this is why it is mandatory to enter the name of the model for the import procedure.

Working with Geometry

This section describes how to set up and run a geometry sequence. In this section:

- The Geometry Sequence Syntax
- Displaying the Geometry
- Working with Geometry Sequences
- Exchanging Geometries with the COMSOL Desktop
- Importing and Exporting Geometries and CAD Models from File
- Retrieving Geometry Information
- Modeling with a Parameterized Geometry
- Images and Interpolation Data

See Also

- Geometry Modeling and CAD Tools in the *COMSOL Multiphysics User's Guide*
- Geometry in the *COMSOL Java API Reference Guide*

Important

The links to features described outside of this user guide do not work in the PDF, only from within the online help.

The Geometry Sequence Syntax

Create a geometry sequence using the syntax

```
model.geom.create(<geomtag>, sdim);
```

where `<geomtag>` is a string used to refer to the geometry. The integer *`sdim`* specifies the space dimension of the geometry and it can be either `0`, `1`, `2`, or `3`.

To add an operation to a geometry sequence, use the syntax

```
model.geom(<geomtag>).feature.create(<ftag>, operation);
```

where <*geomtag*> is the string defined when the geometry is created. The string <*ftag*> is used to refer to the operation.

Note

For a list of geometry operations, see About Geometry Commands in the *COMSOL Java API Reference Guide*.

To set the feature property with different values than the default, use the `set` method

```
model.geom(<geomtag>).feature(<ftag>).set(property, <value>);
```

where <*ftag*> is the string defined when creating the operation.

Note

For a property list available for the geometry features see Geometry in the *COMSOL Java API Reference Guide*.

To build the geometry sequence, enter:

```
model.geom(<geomtag>).run;
```

Alternatively, to build the geometry sequence up to a given feature *ftag* enter:

```
model.geom(<geomtag>).run(<ftag>);
```

Displaying the Geometry

Use the function `mphgeom` to display the geometry in a MATLAB figure:

```
mphgeom(model);
```

To specify the geometry to display, enter:

```
mphgeom(model, <geomtag>);
```

When running `mphgeom` the geometry node is automatically built. Set the `build` property to specify how the geometry node is supposed to be built before displaying it. Enter

```
mphgeom(model, <geomtag>, 'build', build);
```

where *build* is a string with the value: `'off'`, `'current'`, or the geometry feature tag <*ftag*>, which, respectively, does not build the geometry (off), builds the geometry

up to the current feature (current), or builds the geometry up to the specified geometry feature node (ftag).

Use the `parent` property to specify the axes handle where to display the plot:

```
mphgeom(model, <geomtag>, 'parent', <axes>);
```

The following property is also available to specify the vertex, edge, or face rendering:

```
edgecolor, edgelabels, edgelabelscolor, edgemode, facealpha,
facelabels, facelabelscolor, facemode, vertexlabels,
vertexlabelscolor, vertexmode.
```

Use `mphgeom` to display a specified geometry entity. To set the geometry entity, enter the `entity` property and set the geometry entity index in the `selection` property to:

```
mphgeom(model, <geomtag>, 'entity', entity, 'selection', <idx>);
```

where *`entity`* can be either `'point'`, `'edge'`, `'boundary'`, or `'domain'`, and `<idx>` is a positive integer array that contains the list of the geometry entity indices.

Working with Geometry Sequences

This section shows how to create geometry sequences using the syntax outlined in The Geometry Sequence Syntax.

CREATING A 1D GEOMETRY

See Also

For more information about 1D geometry modeling, see Creating a 1D Geometry Model in the *COMSOL Multiphysics User's Guide*.

From the MATLAB command prompt, create a 1D geometry model by adding a geometry sequence and then adding geometry features. The last step is to run the sequence using the `run` method.

First create a model object:

```
model = ModelUtil.create('Model');
```

Then continue with the commands:

```
geom1 = model.geom.create('geom1',1);

i1=geom1.feature.create('i1','Interval');
i1.set('intervals','many');
```

```
i1.set('p','0,1,2');

geom1.run;
```

This creates a geometry sequence with a 1D solid object consisting of vertices at $x = 0$, 1, and 2, and edges joining the vertices adjacent in the coordinate list.

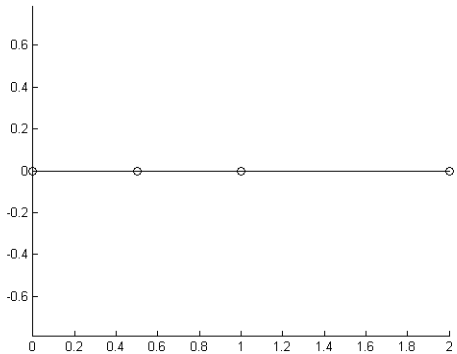
Then enter

```
p1=geom1.feature.create('p1','Point');
p1.set('p',0.5);

geom1.run;
```

to add a point object located at $x = 0.5$ to the geometry.

To plot the result, enter:

```
mphgeom(model,'geom1','vertexmode','on')
```



CREATING A 2D GEOMETRY USING PRIMITIVE GEOMETRY OBJECTS

See Also

For more information about 2D geometry modeling, see Creating a 2D Geometry Model in the *COMSOL Multiphysics User's Guide*.

Creating Composite Objects

Use a model object with a 2D geometry. Enter:

```
model = ModelUtil.create('Model');
```

```
geom2 = model.geom.create('geom2',2);
```

Continue by creating a rectangle with side length of 2 and centered at the origin:

```
sq1 = geom2.feature.create('sq1','Square');
sq1.set('size',2);
sq1.set('base','center');
```

The property `size` describes the side lengths of the rectangle and the property `pos` describes the positioning. The default is to position the rectangle about its lower left corner. Use the property `base` to control the positioning.

Create a circular hole with a radius of 0.5 centered at (0, 0):

```
c1 = geom2.feature.create('c1','Circle');
c1.set('r',0.5);
c1.set('pos',[0 0]);
```

The property `r` describes the radius of the circle, and the property `pos` describes the positioning. The property `pos` could have been excluded because the default position is the origin. The default is to position the circle about its center.

Drill a hole in the rectangle by subtracting the circle from it:

```
co1 = geom2.feature.create('co1','Compose');
co1.selection('input').set({'c1' 'sq1'});
co1.set('formula','sq1-c1');
```

A selection object is used to refer to the input object. The operators `+`, `*`, and `-` correspond to the set operations union, intersection, and difference, respectively.

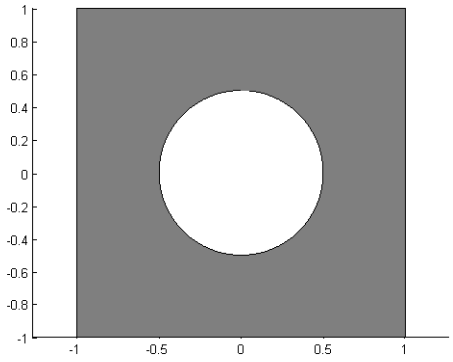
The `Compose` operation allows you to work with a formula. Alternatively use the `Difference` operation instead of `Compose`. The following sequence of commands starts with disabling the `Compose` operation:

```
co1.active(false)

dif1 = geom2.feature.create('dif1','Difference');
dif1.selection('input').set({'sq1'});
dif1.selection('input2').set({'c1'});
```

Run the geometry sequence to create the geometry and plot the result:

```
geom2.run;
```

```
mphgeom(model,'geom2');
```



Trimming Solids

Continue with rounding the corners of the rectangle with the `Fillet` operation:

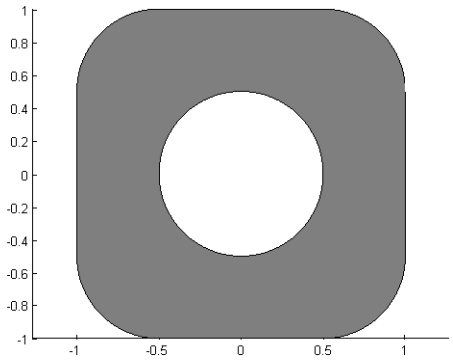
```
fil1 = geom2.feature.create('fil1','Fillet');
fil1.selection('point').set('dif1', [1 2 7 8]);
fil1.set('radius','0.5');
```

Run the sequence again:

```
geom2.run;
```

The geometry sequence is updated with rounded corners. To view the result, enter:

```
mphgeom(model,'geom2');
```



CREATING A 2D GEOMETRY USING BOUNDARY MODELING

Use the following commands to create six open curve segments that together form a closed curve:

```
model = ModelUtil.create('Model');

g1 = model.geom.create('g1',2);

w=1/sqrt(2);

c1 = g1.feature.create('c1','BezierPolygon');
c1.set('type','open');
c1.set('degree',2);
c1.set('p',[-0.5 -1 -1;-0.5 -0.5 0]);
c1.set('w',[1 w 1]);

c2 = g1.feature.create('c2','BezierPolygon');
c2.set('type','open');
c2.set('degree',2);
c2.set('p',[-1 -1 -0.5;0 0.5 0.5]);
c2.set('w',[1 w 1]);

c3 = g1.feature.create('c3','BezierPolygon');
c3.set('type','open');
c3.set('degree',1);
c3.set('p',[-0.5 0.5; 0.5 0.5]);

c4 = g1.feature.create('c4','BezierPolygon');
c4.set('type','open');
c4.set('degree',2);
c4.set('p',[0.5 1 1; 0.5 0.5 0]);
c4.set('w',[1 w 1]);

c5 = g1.feature.create('c5','BezierPolygon');
c5.set('type','open');
c5.set('degree',2);
c5.set('p',[1 1 0.5; 0 -0.5 -0.5]);
c5.set('w',[1 w 1]);

c6 = g1.feature.create('c6','BezierPolygon');
c6.set('type','open');
c6.set('degree',1);
c6.set('p',[0.5 -0.5; -0.5 -0.5]);
```

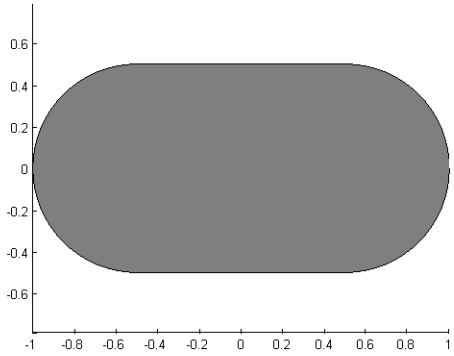
The objects `c1`, `c2`, `c3`, `c4`, `c5`, and `c6` are all `curve2` objects. The vector `[1 w 1]` specifies the weights for a rational Bézier curve that is equivalent to a quarter-circle arc. The weights can be adjusted to create elliptical or circular arcs.

Convert the curve segments to a solid with the following conversion command:

```
csol1 = g1.feature.create('csol1','ConvertToSolid');
csol1.selection('input').set({'c1' 'c2' 'c3' 'c4' 'c5' 'c6'});
```

Then issue a final `run` command:

```
g1.run;
mphgeom(model,'g1');
```



CREATING A 3D GEOMETRY USING SOLID MODELING

See Also

For more information about 3D geometry modeling, see Creating a 3D Geometry Model in the *COMSOL Multiphysics User's Guide*.

This section shows how to create 3D solids using workplanes and Boolean operations.

Create a 3D geometry with an *xy* work plane at $z = 0$:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1', 3);

wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
```

Add a rectangle to the work plane, then add fillet to its corners:

```
r1 = wp1.geom.feature.create('r1', 'Rectangle');
r1.set('size',[1 2]);

geom1.run
```

```
fil1 = wp1.geom.feature.create('fil1', 'Fillet');
fil1.selection('point').set('r1', [1 2 3 4]);
fil1.set('radius', '0.125');

geom1.runCurrent;

ext1 = geom1.feature.create('ext1', 'Extrude');
ext1.set('distance', '0.1');
```

Add another *yz* work plane, at $x = 0.5$:

```
wp2 = geom1.feature.create('wp2', 'WorkPlane');
wp2.set('planetype', 'quick');
wp2.set('quickplane', 'yz');
wp2.set('quickx', '0.5');

b1 = wp2.geom.feature.create('b1', 'BezierPolygon');
b1.set('type', 'open');
b1.set('degree', [1 1 1 1]);
b1.set('p',
{'0.75','1','1','0.8','0.75';'0.1','0.1','0.05','0.05','0.1'});
b1.set('w', {'1','1','1','1','1','1','1','1'});

wp2.geom.feature.create('csol1', 'ConvertToSolid');
wp2.geom.feature('csol1').selection('input').set({'b1'});
```

Revolve the triangle from the *yz* work plane:

```
rev1 = geom1.feature.create('rev1', 'Revolve');
rev1.selection('input').set({'wp2'});
rev1.setIndex('pos', '1', 0);
```

Add the difference operation that computes the final 3D geometry.

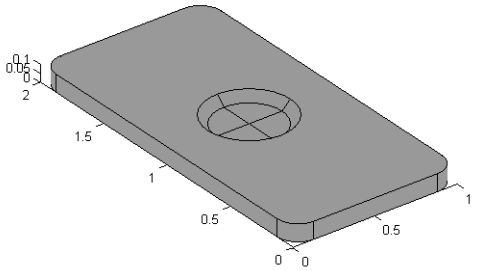
```
dif1 = geom1.feature.create('dif1', 'Difference');
dif1.selection('input').set({'ext1'});
dif1.selection('input2').set({'rev1'});
```

To run the sequence, enter:

```
model.geom('geom1').run;
```

To view the geometry enter:

```
mphgeom(model);
```



Exchanging Geometries with the COMSOL Desktop

EXCHANGING A GEOMETRY FROM THE COMSOL DESKTOP

To transfer a geometry from the COMSOL Desktop to the LiveLink interface in MATLAB, use one of these methods.

Export the Geometry from the COMSOL Desktop

Export the geometry as a COMSOL Multiphysics binary (`.mphbin`) file from the COMSOL Desktop. Right-click the **Geometry** node and select **Export to File**. Then create a geometry import feature from MATLAB:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1', 3);

imp1 = geom1.feature.create('imp1','Import');
imp1.set('filename','geometryfile.mphbin');
imp1.importData;

geom1.run;
```

Save the Geometry

Save the model containing the geometry sequence from the COMSOL Desktop. Create a model object from MATLAB and load the file into it.

Export the Model to the COMSOL Server

Export the model containing the geometry sequence to the COMSOL server.

Importing and Exporting Geometries and CAD Models from File

With COMSOL Multiphysics, you can import and export geometries in a variety of file formats.

COMSOL MULTIPHYSICS FILES

A natural choice for storing geometries in 1D, 2D, and 3D is the native file format of COMSOL's geometry kernel (`.mphtxt` and `.mphbin`).

Note

The `.mphtxt` or `.mphbin` file formats are only used for geometry and mesh objects. It is not the same as a Model MPH-file (`.mph`).

2D CAD FORMATS

COMSOL Multiphysics supports import and export for the *DXF*® file format, a data interchange format of the CAD system AutoCAD®. Files can also be imported using the neutral GDS format. The ECAD geometry file format requires either the AC/DC Module, MEMS Module, or the RF Module.

Tip

See the *ECAD Import Module User's Guide* or go to http://www.comsol.com/products/multiphysics/ for more information about this and other products.

3D CAD FORMATS

It is possible to import surface meshes in the STL and VRML formats. With a license for the CAD Import Module, or one of the LiveLink products for CAD, you can import most 3D CAD file formats: Parasolid®, ACIS® (SAT®), STEP, IGES, Pro/ENGINEER®, Autodesk Inventor®, and SolidWorks®. See the individual user guides for detailed information.

Retrieving Geometry Information

Note

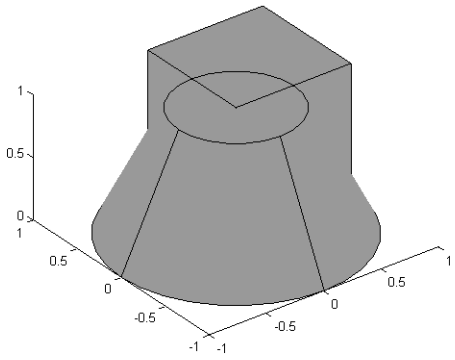
To retrieve the detailed information about the geometry in a model, see Geometry Object Information in the *COMSOL Java API Reference Guide*.

First create a simple 3D geometry:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1', 3);
geom1.feature.create('blk1','Block');
geom1.feature.create('con1','Cone');
geom1.run;
```

To visualize the geometry in a MATLAB figure window enter:

```
mphgeom(model)
```



The model object contains general geometry information methods. For example to determine the space dimension of the geometry, enter:

```
geom1.getSDim
```

There are also methods to determine the number of geometrical entities. For example, to inquire about the number of domains and the number of boundaries:

```
geom1.getNDomains
geom1.getNBoundaries
```

Another group of geometry information methods concern adjacency properties of the geometric entities. For example, the number of up and down domain information on each boundary:

```
geom1.getUpDown
```

There are also methods for evaluating properties such as coordinate values and curvatures on faces and edges. For example, to evaluate coordinates on face 1 for the face parameters (2, 0.005), enter:

```
geom1.faceX(1,[2,0.005])
```

To get the parameters of a given face, use the method `faceParamRange(N)`, where `N` is the face number. For example

```
geom1.faceParamRange(1)
```

returns the parameters for face 1.

To get the parameter range of an edge, use the `edgeParamRange(N)` method. For example to get the length of edge number `3`, enter:

```
geom1.edgeParamRange(3)
```

To get the coordinate and the curvature data along a specified edge, enter:

```
geom1.edgeX(2,0.5)
geom1.edgeCurvature(2,0.5)
```

There are also methods for getting information about the internal representation of the geometry. For example, the coordinates of the geometry vertices:

```
geom1.getVertexCoord
```

To fetch geometry information from elements in the geometry sequence, enter:

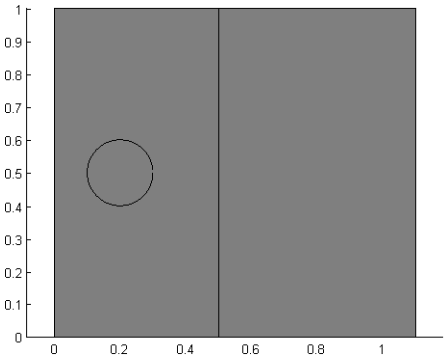
```
geom1.obj('blk1').getNBoundaries
```

Modeling with a Parameterized Geometry

COMSOL has built-in support for parameterized geometries. Parameters can be used in most geometry operations. To exemplify parameterizing a geometry, the following script studies the movement of a circular source through two adjacent rectangular domains:

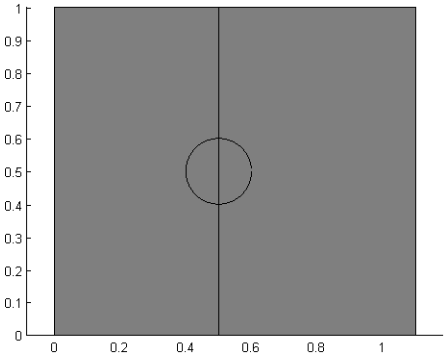
```
model = ModelUtil.create('Model');
model.param.set('a','0.2');

geom1 = model.geom.create('geom1',2);

r1 = geom1.feature.create('r1','Rectangle');
r1.set('size',[0.5 1]);
r1.set('pos',[0 0]);

r2 = geom1.feature.create('r2','Rectangle');
r2.set('size',[0.6 1]);
r2.set('pos',[0.5 0]);

c1 = geom1.feature.create('c1','Circle');
c1.set('r',0.1);
```

```
c1.set('pos',{'a','0.5'});

geom1.run;
mphgeom(model);
```



Change the position of the circle by changing the value of parameter `a`:

```
model.param.set('a','0.5');

geom1.run;

mphgeom(model);
```



Create a loop that changes the position of the circle in increments:

```
for a=0.2:0.1:0.5
  model.param.set('a',a);
  geom1.run;
```

```
end
```

Create a mesh:

```
model.mesh.create('mesh1', 'geom1');
```

Add a Weak Form PDE interface:

```
w = model.physics.create('w', 'WeakFormPDE', 'geom1');
w.feature('wfeq1').set('weak', 1, '-test(ux)*ux-test(uy)*uy');

dir1 = w.feature.create('dir1', 'DirichletBoundary', 1);
dir1.selection.set([1 2 3 6 7]);

src1 = w.feature.create('src1', 'SourceTerm', 2);
src1.set('f', 1, '1');
src1.selection.set([3]);
```

Then, create a stationary study step:

```
std1 = model.study.create('std1');

stat1 = std1.feature.create('stat1', 'Stationary');
```

Create a parametric sweep feature:

```
p1 = model.batch.create('p1','Parametric');
p1.set('pname', 'a');
p1.set('plist','range(0.2,0.1,0.8)');
p1.run;
```

Alternatively, you can run the parametric sweep using a MATLAB for loop:

```
for a=0.2:0.1:0.8
  model.param.set('a',a);
  std1.run;
end
```

Note

After updating a parameter that affects the geometry, COMSOL detects this change and automatically updates the geometry and mesh before starting the solver. The geometry is associative, which means that physics settings are preserved as the geometry changes.

Images and Interpolation Data

This section describes how to generate geometry from a set of data points by using interpolation curves and how to create geometry from image data.

CREATING A GEOMETRY USING CURVE INTERPOLATION

Use the interpolation spline feature to import a set of data points that describe a 2D geometry. To create an interpolation spline feature, enter:

```
model.geom(<geomtag>).feature.create(<ftag>,'InterpolationCurve')
```

Then specify data points in a table

```
model.geom(<geomtag>).feature(<ftag>).set('table',<data>)
```

where `<data>` can either be a 2xN cell array or a 2xN array.

Control the type of geometry generated by the operation with the command

```
model.geom(<geomtag>).feature(<ftag>).set('type',type)
```

where *type* can either be `'solid'` to generate a solid object, `'closed'` to generate a closed curve or `'open'` to generate an open curve.

Example: Curve Interpolation

Create a set of data points in MATLAB, then use these to construct a 2D geometry.

1. Create data points that describe a circle, sorted by the angle, and remove some of the points:

```
phi = 0:0.2:2*pi;
phi([1 3 6 7 10 20 21 25 28 32]) = [];
p = [cos(phi);sin(phi)];
```

2. Add some noise to the data points:

```
randn('state',17)
p = p+0.02*randn(size(p));
```

3. Create a 2D geometry with a square:

```
model = ModelUtil.create('Model');
```

4. Add a square geometry:

```
geom1 = model.geom.create('geom1', 2);

sq1 = geom1.feature.create('sq1', 'Square');
sq1.set('base', 'center');
sq1.set('size', '3');
```

5. Add an interpolation curve feature:

```
ic1 = geom1.feature.create('ic1', 'InterpolationCurve');
```

6. Use the variable `p` for the data points:

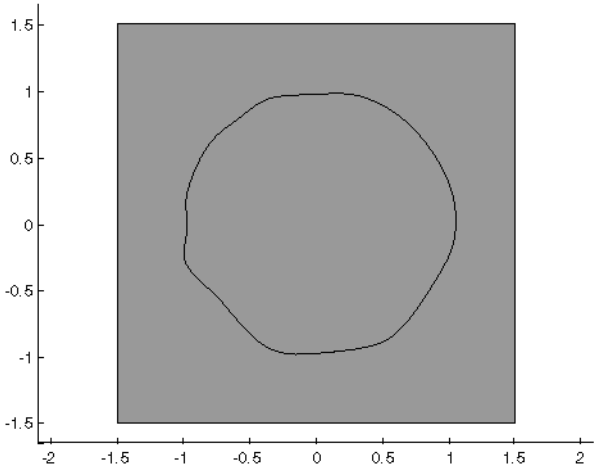
```
ic1.set('table', p');
```

7 Specify a closed curve:

```
ic1.set('type', 'closed');
```

8 Plot the geometry with the `mphgeom` command:

```
mphgeom(model);
```



CREATING GEOMETRY FROM IMAGE DATA

Use the function `mphimage2geom` to create geometry from image data. The image data format can be *M*-by-*N* array for a grayscale image or *M*-by-*N*-by-3 array for a true color image.

Note

See the MATLAB function `imread` to convert an image file to image data.

If you specify the image data and the level value that represents the geometry contour you want to extract, the function `mphimage2geom` returns a model object with the desired geometry.

```
model = mphimage2geom(<imagedata>, <level>)
```

where *`imagedata`* is a C array containing the image data and *`level`* is the contour level value used to generate the geometry contour.

Specify the type of geometry object generated

```
model = mphimage2geom(<imagedata>, <level>, 'type', type)
```

where *type* is `'solid'` and generates a solid object, `'closed'` generates a closed curve object, or `'open'` generates an open curve geometry object.

Use the property `curvetype` to specify the type of curve used to generate the geometry object

```
model = mphimage2geom(<imagedata>, <level>, 'curvetype', curvetype)
```

where *curvetype* can be set to `'polygon'` to use a polygon curve. The default curve type creates a geometry with the best suited geometrical primitives. For interior curves it uses *interpolation curves*, while for curves that are touching the perimeter of the image a *polygon curve* is used.

To `scale` the geometry use the scale property where *scale* is a double value:

```
model = mphimage2geom(<imagedata>, <level>, 'scale', scale)
```

Set the minimum distance between coordinates in curve with the `mindist` property where *mindist* is a double value:

```
model = mphimage2geom(<imagedata>, <level>, 'mindist', mindist)
```

Set the minimum area for interior curves where *minarea* is a double value:

```
model = mphimage2geom(<imagedata>, <level>, 'minarea', minarea)
```

In case of overlapping solids, the function `mphimage2geom` automatically creates a `Compose` node in the model object. If you do not want this geometry feature, set the property `compose` to `off`:

```
model = mphimage2geom(<imagedata>, <level>, 'compose', 'off')
```

To create a rectangle domain surrounding the object generated use the property `rectangle`:

```
model = mphimage2geom(<imagedata>, <level>, 'rectangle', 'on')
```

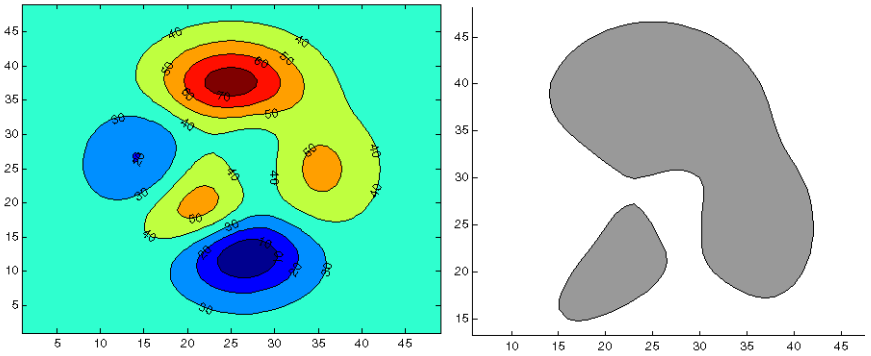
Example: Convert Image Data to Geometry

This example illustrates how to create geometry based on gray scale image data. First generate the image data in MATLAB and display the contour in a figure. Then, create a model object including the geometry represented by the contour value 40.

At the MATLAB prompt enter these commands:

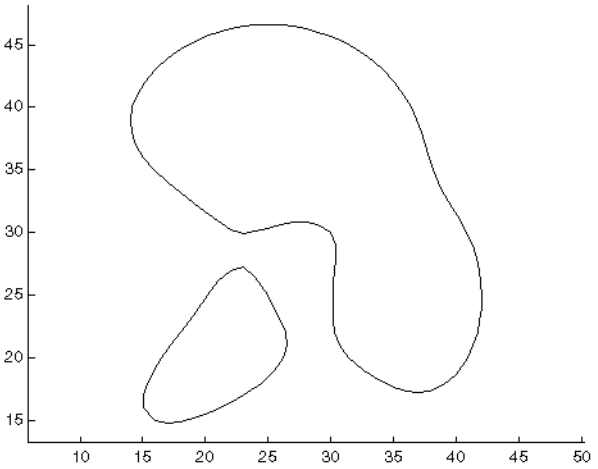
```
p = (peaks+7)*5;
[c,h] = contourf(p);
```

```
clabel(c, h);
model = mphimage2geom(p, 40);
figure(2)
mphgeom(model)
```



Use the property `type` to create closed or open curves. For example, to create a geometry following contour 40 with closed curves, enter:

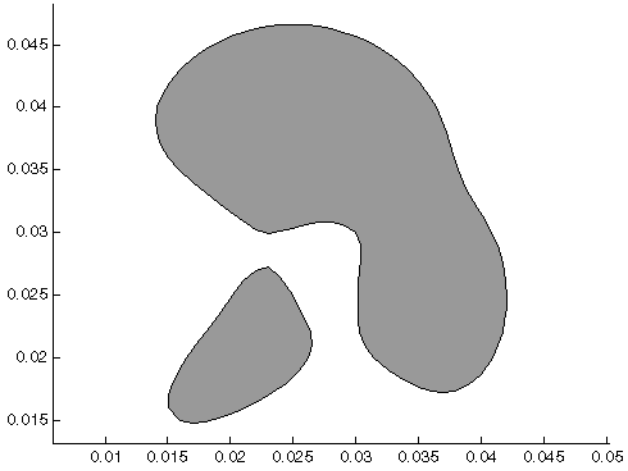
```
model = mphimage2geom(p, 40, 'type', 'closed');
mphgeom(model)
```



To scale the geometry, use the `scale` property. Using the current model scale the geometry with a factor of 0.001 (1e-3):

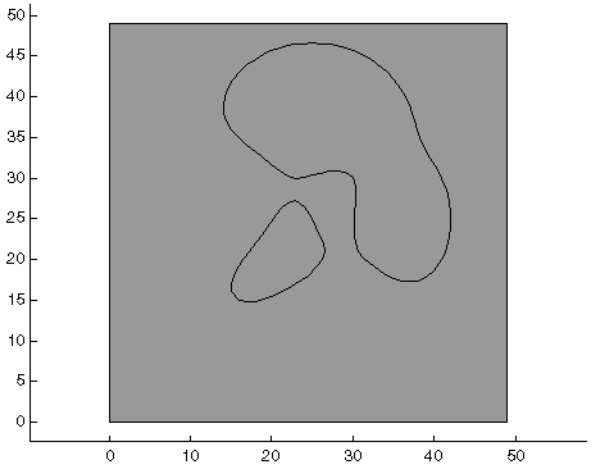
```
model = mphimage2geom(p, 40, 'scale', 1e-3);
```

```
mphgeom(model)
```



To insert a rectangle in the geometry that has an outer domain surrounding the created contour, set the property `rectangle` to `on`:

```
model = mphimage2geom(p, 40, 'rectangle', 'on');
mphgeom(model)
```



Working with Meshes

This section describes how to set up and run meshing sequences in a model.

- The Meshing Sequence Syntax
- Displaying the Mesh
- Mesh Creation Functions
- Importing External Meshes and Mesh Objects
- Measuring Mesh Quality
- Getting Mesh Statistics Information
- Getting and Setting Mesh Data

See Also

- Creating Meshes in the *COMSOL Multiphysics User's Guide*
- Mesh in the *COMSOL Java API Reference Guide*

Important

The links to features described outside of this user guide do not work in the PDF, only from within the online help.

The Meshing Sequence Syntax

Create a meshing sequence by using the syntax

```
model.mesh.create(<meshtag>, <geomtag>);
```

where *<meshtag>* is a string that you use to refer to the sequence. The tag *geomtag* specifies the geometry to use for this mesh node.

To add an operation to a sequence, use the syntax

```
model.mesh(<meshtag>).feature.create(<ftag>, operation);
```

where the string <*ftag*> is a string that you use to refer to the operation.

See Also

See About Mesh Commands in the *COMSOL Java API Reference Guide*.

To set a property to a value in a operation, enter:

```
model.mesh(<meshtag>).feature(<ftag>).set(property, <value>);
```

To build the mesh sequence, enter:

```
model.mesh(<meshtag>).run;
```

Or to run the mesh node up to a specified feature node <*ftag*>, enter:

```
model.mesh(<meshtag>).run(ftag);
```

See Also

For more details on available operations and properties in the sequence, see Mesh in the *COMSOL Java API Reference Guide*.

Displaying the Mesh

To display the mesh in a MATLAB figure, use the function `mphmesh`. Make sure that the mesh is built before calling this command:

```
mphmesh(model);
```

If you have several meshes in your model, specify the mesh to display using the command:

```
mphmesh(model, <meshtag>);
```

Use the `parent` property to specify the axes handle where to display the plot:

```
mphmesh(model, <meshtag>, 'parent', <axes>);
```

The following properties are also available to specify the vertex, edge or face rendering: `edgecolor`, `edgelabels`, `edgelabelscolor`, `edgemode`, `facealpha`, `facelabels`, `facelabelscolor`, `facemode`, `meshcolor`, `vertexlabels`, `vertexlabelscolor`, `vertexmode`.

MESH SIZING PROPERTIES

The `Size` attribute provides a number of input properties that can control the mesh element size, such as the:

- maximum and minimum element size
- element growth rate
- resolution of curvature
- resolution of narrow regions

These properties are available both globally and locally.

There are several predefined settings that can be used to set a suitable combination of values for many properties. To select one of these settings, use the property `hauto` and pass an integer from 1 to 9 as its value to describe the mesh resolution:

- Extremely fine (1)
- Extra fine (2)
- Finer (3)
- Fine (4)
- Normal (5)
- Coarse (6)
- Coarser (7)
- Extra coarse (8)
- Extremely coarse (9)

The default mesh setting is `Normal (5)`.

See Also

For details about predefined mesh size settings and mesh element size parameters, see `Size` in the *COMSOL Java API Reference Guide*.

Example: Creating a 2D Mesh with Triangular Elements

Generate a triangular mesh of a unit square:

```
model = ModelUtil.create('Model');
```

```
geom1 = model.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');

mesh1 = model.mesh.create('mesh1','geom1');
ftri1 = mesh1.feature.create('ftri1','FreeTri');
mesh1.run;

mphmesh(model);
```

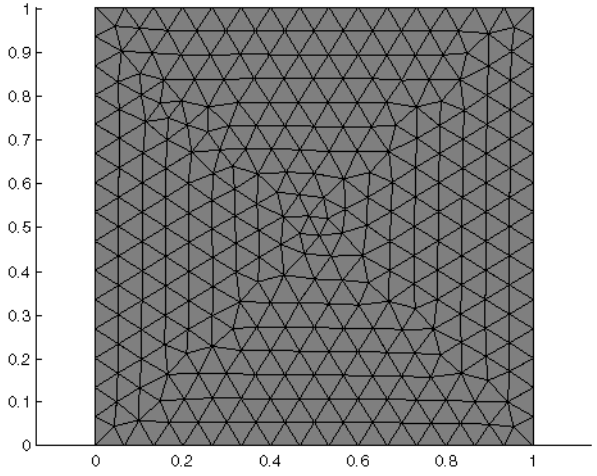


Figure 3-1: Default mesh on a unit square.

The default size feature is generated with the property `hauto` set to `5`, that is:

```
mesh1.feature('size').set('hauto','5');
```

To override this behavior, set `hauto` to another integer. Override this by setting specific size properties, for example, making the mesh finer than the default by specifying a maximum element size of 0.02:

```
mesh1.feature('size').set('hmax','0.02');
mesh1.run;

mphmesh(model);
```

This value corresponds to 1/50 of the largest axis-parallel distance, whereas the default value is 1/15.

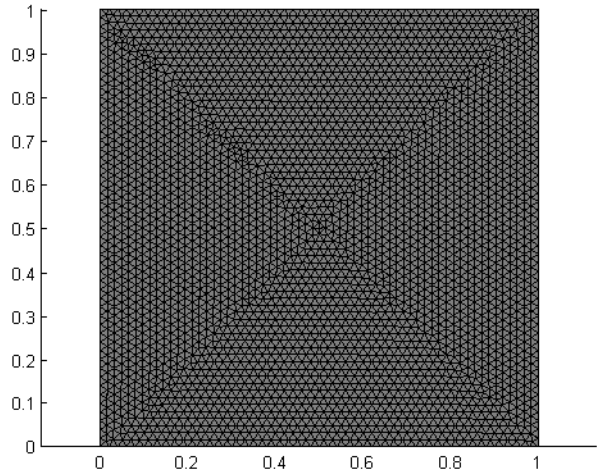


Figure 3-2: Fine mesh (maximum element size = 0.02).

Sometimes a nonuniform mesh is desirable. Make a mesh that is denser on the left side by specifying a smaller maximum element size only on the edge segment to the left (edge number 1):

```
mesh1.feature('size').set('hauto','5');

size1 = ftri1.feature.create('size1','Size');
size1.set('hmax','0.02');
size1.selection.geom('geom1',1);
size1.selection.set(1);

mesh1.run

mphmesh(model);
```

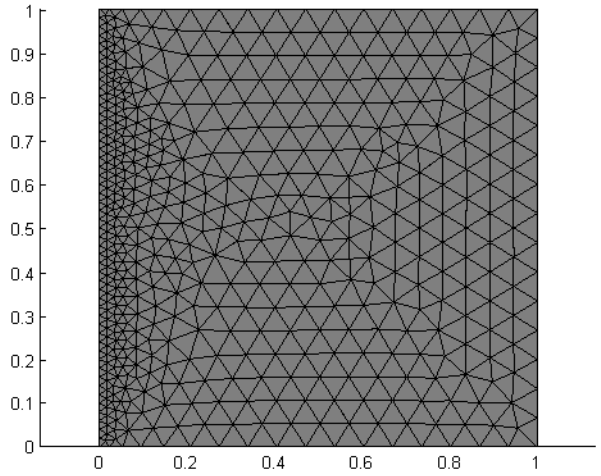


Figure 3-3: Nonuniform mesh.

The Free Meshing Method

The default method to generate free triangle meshes in 2D is based on an advancing front algorithm. To switch to a Delaunay algorithm use the value `del` for the `method` property. Start by creating a geometry:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1',2);

geom1.feature.create('r1','Rectangle');

c1 = geom1.feature.create('c1','Circle');
c1.set('r','0.5');

co1=geom1.feature.create('co1','Compose');
co1.selection('input').set({'c1' 'r1'});
co1.set('formula','r1-c1');

geom1.runAll;

mesh1 = model.mesh.create('mesh1','geom1');

ftri1 = mesh1.feature.create('ftri1','FreeTri');
ftri1.set('method','del');
```

```
mesh1.run;

mphmesh(model,'mesh1')
```

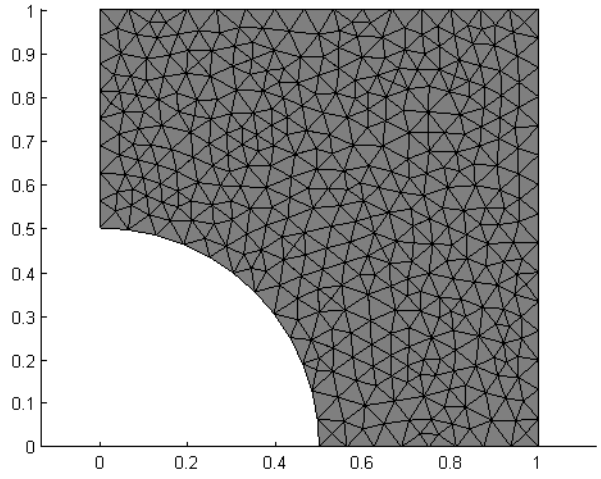


Figure 3-4: Mesh created with the Delaunay method.

Example: Creating a 2D Mesh with Quadrilateral Elements

To create an unstructured quadrilateral mesh on a unit circle, enter:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1',2);

geom1.feature.create('c1','Circle');

mesh1 = model.mesh.create('mesh1','geom1');

mesh1.feature.create('ftri1','FreeQuad');

mesh1.run;

mphmesh(model)
```

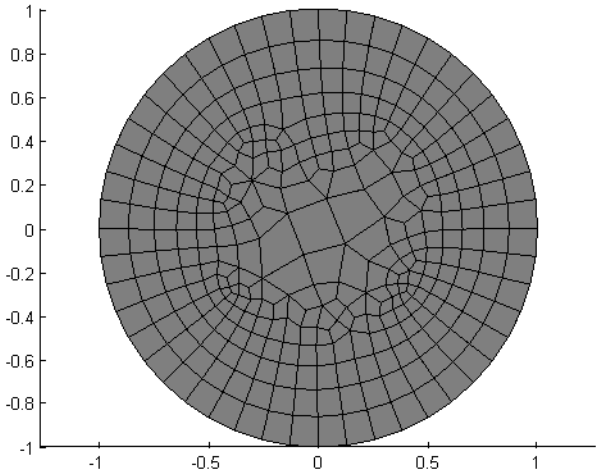


Figure 3-5: Free quad mesh.

CREATING STRUCTURED MESHES

To create a structured quadrilateral mesh in 2D, use the `Map` operation. This operation uses a mapping technique to create the quadrilateral mesh.

See Also

`Map` in the *COMSOL Java API Reference Guide*

Use the `EdgeGroup` attribute to group the edges (boundaries) into four edge groups, one for each edge of the logical mesh. To control the edge element distribution use the `Distribution` attribute, which determines the overall mesh density.

Example: Creating a Structured Quadrilateral Mesh

Create a structured quadrilateral mesh on a geometry where the domains are bounded by more than four edges:

```
model = ModelUtil.create('Model');
geom1 = model.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');
r2 = geom1.feature.create('r2','Rectangle');
```

```
r2.set('pos',[1 0]);
c1 = geom1.feature.create('c1','Circle');
c1.set('r','0.5');
c1.set('pos',[1.1 -0.1]);
dif1 = geom1.feature.create('dif1', 'Difference');
dif1.selection('input').set({'r1' 'r2'});
dif1.selection('input2').set({'c1'});
geom1.run('dif1');

mesh1 = model.mesh.create('mesh1','geom1');

map1 = mesh1.feature.create('map1','Map');

eg1 = map1.feature.create('eg1', 'EdgeGroup');
eg1.selection.set(1);
eg1.selection('edge1').set([1 3]);
eg1.selection('edge2').set(2);
eg1.selection('edge3').set(8);
eg1.selection('edge4').set(4);

eg2 = map1.feature.create('eg2', 'EdgeGroup');
eg2.selection.set(2);
eg2.selection('edge1').set(4);
eg2.selection('edge2').set([6 9 10]);
eg2.selection('edge3').set(7);
eg2.selection('edge4').set(5);

mesh1.run;
mphmesh(model);
```

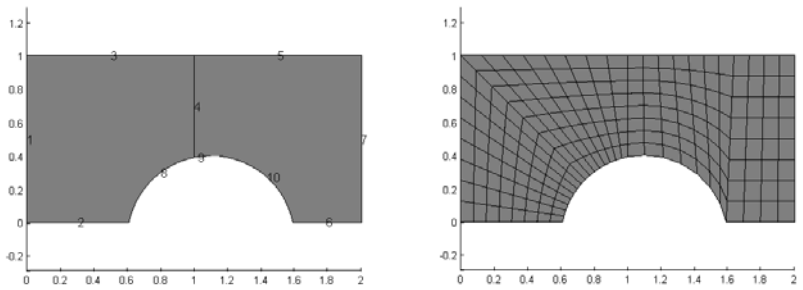


Figure 3-6: Structured quadrilateral mesh (right) and its underlying geometry.

The left-hand side plot in Figure 3-6 is obtained with this command:

```
mphgeom(model, 'geom1', 'edgelabels','on')
```

The `EdgeGroup` attributes specify that the four edges enclosing domain 1 are boundaries 1 and 3; boundary 2; boundary 8; and boundary 4. For domain 2 the four edges are boundary 4; boundary 5; boundary 7; and boundaries 9, 10, and 6.

BUILDING A MESH INCREMENTALLY

To build meshes in a step-by-step fashion, create selections for the parts of the geometry that you want to mesh in each step, as in this example:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');
geom1.feature.create('c1','Circle');
uni1 = geom1.feature.create('uni1', 'Union');
uni1.selection('input').set({'c1' 'r1'});
geom1.runCurrent;
del1 = geom1.feature.create('del1', 'Delete');
del1.selection('input').init(1);
del1.selection('input').set('uni1', 8);
geom1.run('del1');

mesh1 = model.mesh.create('mesh1','geom1');

dis1 = mesh1.feature.create('dis1', 'Distribution');
dis1.selection.set([2 4]);
dis1.set('type', 'predefined');
dis1.set('method', 'geometric');
dis1.set('elemcount', '20');
dis1.set('reverse', 'on');
dis1.set('elemratio', '20');

dis2 = mesh1.feature.create('dis2', 'Distribution');
dis2.selection.set([1 3]);
dis2.set('type', 'predefined');
dis2.set('method', 'geometric');
dis2.set('elemcount', '20');
dis2.set('elemratio', '20');

map1 = mesh1.feature.create('map1','Map');
map1.selection.geom('geom1', 2);
map1.selection.set(2);
mesh1.feature.create('frt1','FreeTri');

mesh1.run;

mphmesh(model);
```

The final mesh is in Figure 3-7. Note the effect of the `Distribution` feature, with which the distribution of vertex elements along geometry edges can be controlled.

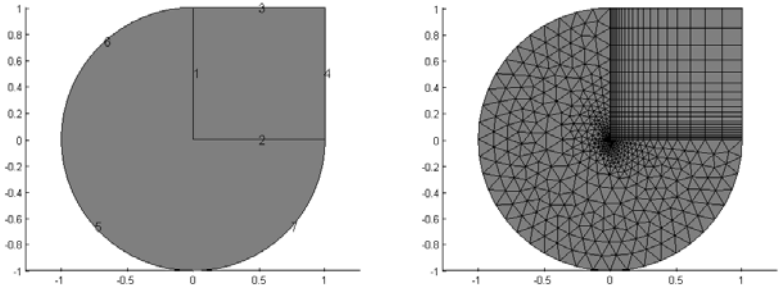


Figure 3-7: Incrementally generated mesh (right).

The left-hand side plot in Figure 3-7 is obtained with this command:

```
mphgeom(model, 'geom1', 'edgelabels','on')
```

To replace the structured quad mesh by an unstructured quad mesh, delete the `Map` feature and replace it by a `FreeQuad` feature:

```
mesh1.feature.remove('map1');
mesh1.run('dis1');
fq1 = mesh1.feature.create('fq1', 'FreeQuad');
fq1.selection.geom('geom1', 2).set(2);
mesh1.run;
```

Analogous to working with the meshing sequence in the Model Builder in the COMSOL Desktop, new features are always inserted after the current feature. Thus, to get the `FreeQuad` feature before the `FreeTri` feature, the `dis1` feature needs to be made the current feature by building it with the `run` method.

Alternatively, parts of a mesh can be selectively removed by using the `Delete` feature. For example, to remove the structured mesh from domain 2 (along with the adjacent edge mesh on edges 3 and 4), and replace it with an unstructured quad mesh, enter these commands:

```
del1 = mesh1.feature.create('del1','Delete');
del1.selection.geom('geom1', 2).set(2);
del1.set('deladj','on');
frq1 = mesh1.feature.create('frq1','FreeQuad');
frq1.selection.geom('geom1', 2).set(2);
mesh1.run;
```

See Also

For further details on the various commands and their properties see the *COMSOL Java API Reference Guide*.

REVOLVING A MESH BY SWEEPING

Create 3D volume meshes by extruding and revolving face meshes with the `Sweep` feature. Depending on the 2D mesh type, the 3D meshes can be hexahedral (brick) meshes or prism meshes.

Example: Revolved Mesh

Create and visualize a revolved prism mesh as follows:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
c1 = wp1.geom.feature.create('c1', 'Circle');
c1.set('pos', [2, 0]);
rev1 = geom1.feature.create('rev1', 'Revolve');
rev1.set('angle2', '60').set('angle1', '-60');
rev1.selection('input').set({'wp1'});
geom1.run('rev1');

mesh1 = model.mesh.create('mesh1', 'geom1');
mesh1.feature.create('ftri1', 'FreeTri');
mesh1.feature('ftri1').selection.geom(2);
mesh1.feature('ftri1').selection.set(2);
mesh1.runCurrent;

swe1 = mesh1.feature.create('swe1', 'Sweep');
swe1.selection.geom(3);
swe1.selection.add(1);

mesh1.run;
mphmesh(model)
```

To obtain a torus, leave the `angles` property unspecified; the default value gives a complete revolution.

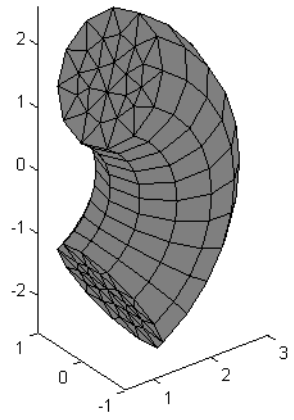


Figure 3-8: 3D prism mesh created with the `Sweep` *feature.*

EXTRUDING A MESH BY SWEEPING

To generate a 3D prism mesh from the same 2D mesh by extrusion and then to plot it, enter these commands:

```
model = ModelUtil.create('Model');
geom1 = model.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
c1 = wp1.geom.feature.create('c1', 'Circle');
c1.set('pos', [2, 0]);
ext1 = geom1.feature.create('ext1', 'Extrude');
ext1.selection('input').set({'wp1'});
geom1.runAll;

mesh1 = model.mesh.create('mesh1', 'geom1');

ftri1 = mesh1.feature.create('ftri1', 'FreeTri');
ftri1.selection.geom('geom1', 2);
ftri1.selection.set(3);

dis1 = mesh1.feature.create('dis1', 'Distribution');
dis1.selection.set(1);
dis1.set('type', 'predefined');
```

```
dis1.set('elemcount', '20');
dis1.set('elemratio', '100');

swe1 = mesh1.feature.create('swe1', 'Sweep');
swe1.selection('sourceface').geom('geom1', 2);
swe1.selection('targetface').geom('geom1', 2);

mesh1.run;
mphmesh(model);
```

The result is shown in Figure 3-9. With the properties `elemcount` and `elemratio` the number and distribution of mesh element layers is controlled in the extruded direction.

`Distribution` in the *COMSOL Java API Reference Guide*

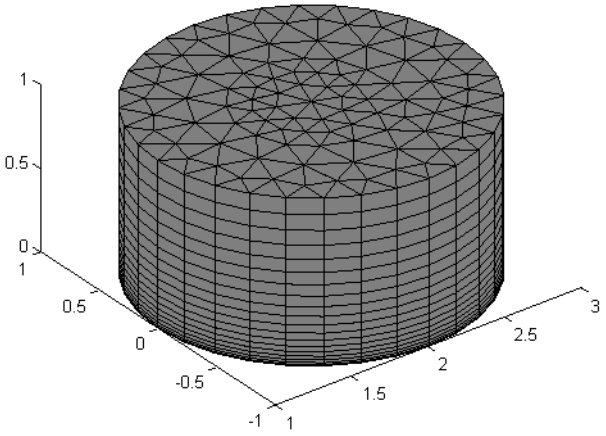


Figure 3-9: Extruded 3D prism mesh.

COMBINING UNSTRUCTURED AND STRUCTURED MESHES

By specifying selections for the meshing operations, swept meshing can also be combined with free meshing. In this case, start by free meshing domain 2, then sweep the resulting surface mesh through domain 1, as in this example:

```
model = ModelUtil.create('Model');
geom1 = model.geom.create('geom1', 3);
cone1 = geom1.feature.create('cone1', 'Cone');
cone1.set('r', '0.3');
cone1.set('h', '1');
cone1.set('ang', '9');
cone1.set('pos', [ 0 0.5 0.5]);
cone1.set('axis', [-1 0 0]);
geom1.feature.create('blk1', 'Block');

mesh1 = model.mesh.create('mesh1', 'geom1');

ftet1 = mesh1.feature.create('ftet1', 'FreeTet');
ftet1.selection.geom('geom1', 3);
ftet1.selection.set(2);

swe1 = mesh1.feature.create('swe1', 'Sweep');
swe1.selection('sourceface').geom('geom1', 2);
swe1.selection('targetface').geom('geom1', 2);

mesh1.run;
mphmesh(model);
```

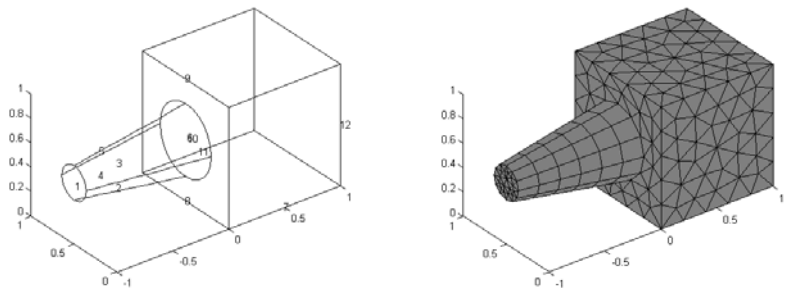


Figure 3-10: Combined structured/unstructured mesh.

The left-hand side plot in Figure 3-10 is obtained with this command:

```
mphgeom(model,'geom1','facemode','off','facelabels','on')
```

CREATING BOUNDARY LAYER MESHES

For 2D and 3D geometries it is also possible to create boundary layer meshes using the `BndLayer` feature. A boundary layer mesh is a mesh with dense element distribution in the normal direction along specific boundaries. This type of mesh is typically used for fluid flow problems to resolve the thin boundary layers along the no-slip boundaries. In 2D, a layered quadrilateral mesh is used along the specified no-slip

boundaries. In 3D, a layered prism mesh or hexahedral mesh is used depending on whether the corresponding boundary layer boundaries contain a triangular or a quadrilateral mesh.

If starting with an empty mesh, the boundary-layer mesh uses free meshing to create the initial mesh before inserting boundary layers into the mesh. This generates a mesh with triangular and quadrilateral elements in 2D and tetrahedral and prism elements in 3D. The following example illustrates this in 2D:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1', 2);
r1 = geom1.feature.create('r1', 'Rectangle');
r1.set('size', [10, 5]);
c1 = geom1.feature.create('c1', 'Circle');
c1.set('pos', [3.5 2.5]);
dif1 = geom1.feature.create('dif1', 'Difference');
dif1.selection('input').set({'r1'});
dif1.selection('input2').set({'c1'});
geom1.runAll;

mesh1 = model.mesh.create('mesh1', 'geom1');

bl1 = mesh1.feature.create('bl1', 'BndLayer');
bl1.feature.create('blp1', 'BndLayerProp');
bl1.feature('blp1').selection.set([2 3 5 6 7 8]);

mesh1.run;
mphmesh(model);
```

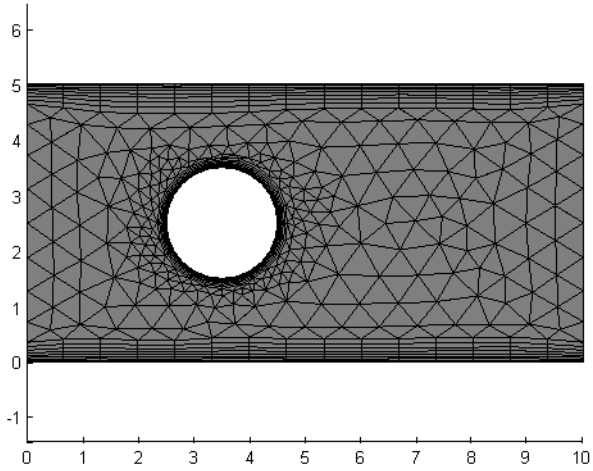


Figure 3-11: Boundary layer mesh based on an unstructured triangular mesh.

It is also possible to insert boundary layers in an existing mesh. Use the following meshing sequence with the geometry sequence from the previous example:

```
bl1.active(false);

fq1 = mesh1.feature.create('fq1', 'FreeQuad');
fq1.selection.set([1]);

mphmesh(model)

bl1 = mesh1.feature.create('bl2', 'BndLayer');
bl1.feature.create('blp2', 'BndLayerProp');
bl1.feature('blp2').selection.set([2 3 5 6 7 8]);
mesh1.run;
```

```
mphmesh(model);
```

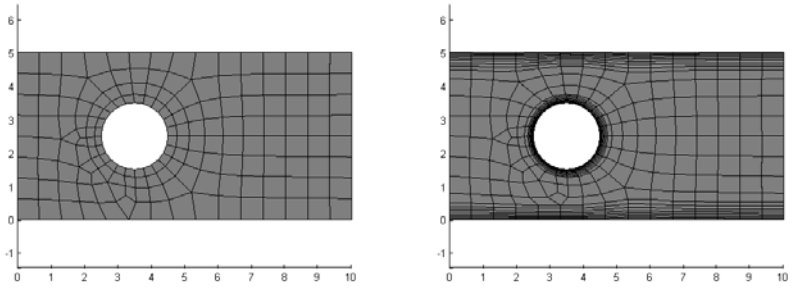


Figure 3-12: Initial unstructured quad mesh (left) and resulting boundary layer mesh (right).

REFINING MESHES

Given a mesh consisting only of *simplex elements* (lines, triangles, and tetrahedra) you can create a finer mesh using the feature `Refine`. Enter this command to refine the mesh:

```
mesh1.feature.create('ref1', 'Refine');
```

By specifying the property `tri`, either as a row vector of element numbers or a 2-row matrix, the elements to be refined can be controlled. In the latter case, the second row of the matrix specifies the number of refinements for the corresponding element.

The refinement method is controlled by the property `rmethod`. In 2D, its default value is `regular`, corresponding to regular refinement, in which each specified triangular element is divided into four triangles of the same shape. Setting `rmethod` to `longest` gives longest edge refinement, where the longest edge of a triangle is bisected. Some triangles outside the specified set might also be refined in order to preserve the triangulation and its quality.

In 3D, the default refinement method is `longest`, while regular refinement is only implemented for uniform refinements. In 1D, the function always uses `regular` refinement, where each element is divided into two elements of the same shape.

Note

For stationary or eigenvalue PDE problems you can use adaptive mesh refinement at the solver stage with the solver step `adaption`. See `Adaption` in the *COMSOL Java API Reference Guide*.

COPYING BOUNDARY MESHES

Use the `CopyEdge` feature in 2D and the `CopyFace` feature in 3D to copy a mesh between boundaries.

Note

It is only possible to copy meshes between boundaries that have the same shape. However, a scaling factor between the boundaries is allowed.

The following example demonstrates how to copy a mesh between two boundaries in 3D and then create a swept mesh on the domain.

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
c1 = wp1.geom.feature.create('c1', 'Circle');
c1.set('r', 0.5);
c1.set('pos', [1, 0]);
rev1 = geom1.feature.create('rev1', 'Revolve');
rev1.set('angle1', '0').set('angle2', '180');
rev1.selection('input').set({'wp1'});
geom1.run('wp1');

mesh1 = model.mesh.create('mesh1', 'geom1');

size1 = mesh1.feature.create('size1', 'Size');
size1.selection.geom('geom1', 1);
size1.selection.set(18);
size1.set('hmax', '0.06');

ftri1 = mesh1.feature.create('ftri1', 'FreeTri');
ftri1.selection.geom('geom1', 2);
ftri1.selection.set(10);

cpf1 = mesh1.feature.create('cpf1', 'CopyFace');
cpf1.selection('source').geom('geom1', 2);
cpf1.selection('destination').geom('geom1', 2);
cpf1.selection('source').set(10);
cpf1.selection('destination').set(1);

sw1 = mesh1.feature.create('sw1', 'Sweep');
sw1.selection('sourceface').geom('geom1', 2);
sw1.selection('targetface').geom('geom1', 2);
```

```
mesh1.run;
mphmesh(model);
```

The algorithm automatically determines how to orient the source mesh on the target boundary. The result is shown in Figure 3-13.

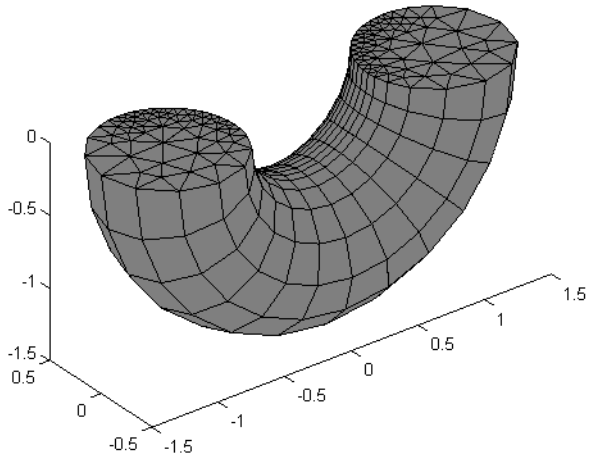


Figure 3-13: Prism element obtained with the `CopyFace` *and* `Sweep` *features.*

To explicitly control the orientation of the copied mesh, use the `EdgeMap` attribute. The command sequence

```
em1 = cpf1.feature.create('em1', 'EdgeMap');
em1.selection('srcedge').set(18);
em1.selection('dstedge').set(2);
mesh1.feature.remove('sw1');
mesh1.feature.create('ftet1', 'FreeTet');

mesh1.run;
mphmesh(model);
```

copies the mesh between the same boundaries as in the previous example, but now the orientation of the source mesh on the target boundary is different. The domain is then meshed by the free mesh, resulting in the mesh in Figure 3-14. In this case it is not possible to create a swept mesh on the domain because the boundary meshes do not match in the sweeping direction.

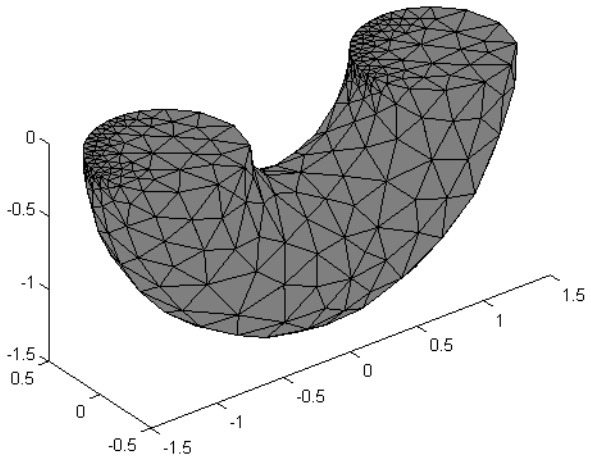


Figure 3-14: Free tetrahedral mesh after the use of the `CopyFace` *feature.*

CONVERTING MESH ELEMENTS

Use the `Convert` feature to convert meshes containing quadrilateral, hexahedral, or prism elements into triangular meshes and tetrahedral meshes. In 2D, the function splits each quadrilateral element into either two or four triangles. In 3D, it converts each prism into three tetrahedral elements and each hexahedral element into five, six, or 28 tetrahedral elements. To control the method used to convert the elements, use the property `splitmethod`. The default value is `diagonal`, which results in two triangular elements in 2D and five or six tetrahedral elements in 3D.

See Also

For additional properties supported, see `Convert` in the *COMSOL Java API Reference Guide*.

This example demonstrates how to convert a quad mesh into a triangle mesh:

```
model = ModelUtil.create('Model');
geom1 = model.geom.create('geom1', 2);
geom1.feature.create('c1', 'Circle');
geom1.feature.create('r1', 'Rectangle');
int1 = geom1.feature.create('int1', 'Intersection');
```

```
int1.selection('input').set({'c1' 'r1'});

mesh1 = model.mesh.create('mesh1', 'geom1');
mesh1.feature.create('fq1', 'FreeQuad');
mesh1.runCurrent;
mesh1.feature.create('conv1', 'Convert');
mesh1.run;
mphmesh(model);
```

The result is illustrated in the Figure 3-15:

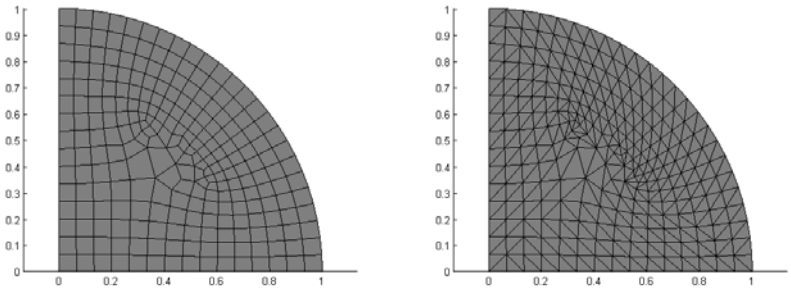


Figure 3-15: Mesh using free quad elements (left) and converted mesh from quad to triangle (right).

Importing External Meshes and Mesh Objects

It is possible to import meshes to COMSOL Multiphysics using the following formats:

- COMSOL Multiphysics text files (extension .mphtxt)
- COMSOL Multiphysics binary files (extension .mphbin)
- NASTRAN files (extension .nas or .bdf)

For a description of the text file format see the *COMSOL Multiphysics Reference Guide*.

IMPORTING MESHES TO THE COMMAND LINE

To import a mesh stored in a supported format use the `Import` feature. The following commands import and plot a NASTRAN mesh for a crankshaft:

```
model = ModelUtil.create('Model');
model.geom.create('geom1', 3);

mesh1 = model.mesh.create('mesh1', 'geom1');
imp1 = mesh1.feature.create('imp1', 'Import');
```

```
model.modelPath('dir\COMSOL43a\models\COMSOL_Multiphysics\
Structural_Mechanics')
imp1.set('filename','crankshaft.nas');
mesh1.feature('imp1').importData;

mesh1.run;
mphmesh(model);
```

Where *dir* is the path of root directory where COMSOL Multiphysics is installed.

The above command sequence results in Figure 3-16.

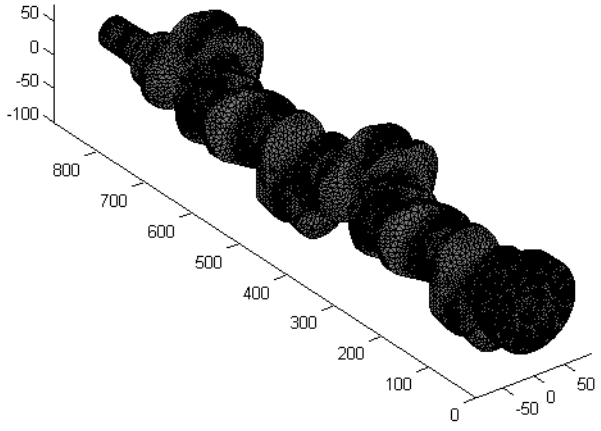


Figure 3-16: Imported NASTRAN mesh.

See Also

For additional properties supported, see `Import` in the *COMSOL Java API Reference Guide*.

Measuring Mesh Quality

Use the `stat` method on the meshing sequence to get information on the mesh quality. The quality measure is a scalar quantity, defined for each mesh element, where 0 represents the lowest quality and 1 represents the highest quality.

The following commands show how to visualize the mesh quality for a mesh on the unit circle:

```
model = ModelUtil.create('Model');
geom1 = model.geom.create('geom1', 2);
geom1.feature.create('c1', 'Circle');
geom1.runAll;

mesh1 = model.mesh.create('mesh1', 'geom1');
mesh1.feature.create('ftri1', 'FreeTri');
mesh1.run;

meshdset1 = model.result.dataset.create('mesh1', 'Mesh');
meshdset1.set('mesh', 'mesh1');

pg1 = model.result.create('pg1', 2);

meshplot1 = pg1.feature.create('mesh1', 'Mesh');
meshplot1.set('data', 'mesh1');
meshplot1.set('filteractive', 'on');
meshplot1.set('elemfilter', 'quality');
meshplot1.set('tetkeep', '0.25');
mphplot(model,'pg1');
meshplot1.set('elemfilter','qualityrev');
meshplot1.run;
mphplot(model,'pg1');
```

These commands display the worst 25% and the best 25% elements in terms of mesh element quality. See how in Figure 3-17 the triangular mesh elements in the plot to the right are more regular than those in the left plot; this reflects the fact that a quality

measure of 1 corresponds to a uniform triangle, while 0 means that the triangle has degenerated into a line.

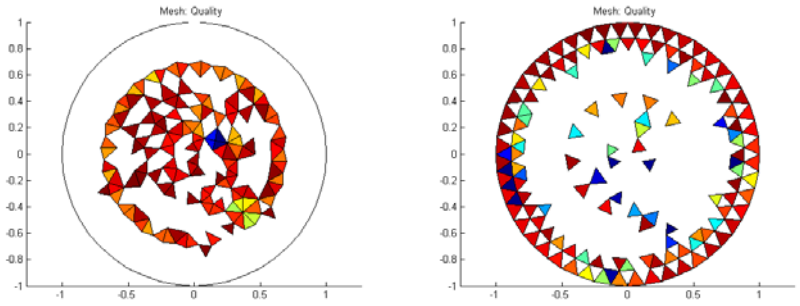


Figure 3-17: Visualizations of the mesh quality: worst 25% (left) and best 25% (right).

Getting Mesh Statistics Information

Use the function `mphmeshstats` to get mesh statistics and mesh information where `stats` is a structure containing the mesh statistics information. Enter:

```
stats = mphmeshstats(model);
```

The statistics structure contains the following fields:

- `meshtag`, the tag of the mesh feature
- `isactive`, Boolean variable that indicates if the mesh feature is active (1) or not (0)
- `hasproblems`, Boolean variable that indicates if the mesh feature contains error or warning nodes (1) or not (0)
- `iscomplete`, Boolean variable that indicates if the mesh feature is built (1) or not(0)
- `sdim`, the space dimension of the mesh feature
- `types`, the element types present in the mesh. The element type can be vertex (`vtx`), edge (`edg`), triangle (`tri`), quadrilateral (`quad`), tetrahedra (`tet`), pyramid (`pyr`), prism (`prism`), hexahedra (`hex`). The type can also be of all elements of maximal dimension in the selection (`all`).
- `numelem`, number of elements for each element type
- `minquality`, minimum element quality
- `meanquality`, mean element quality
- `qualitydistr`, distribution of the element quality (20 values)
- `minvolume`, minimum element volume/area

- `maxvolume`, maximum element volume/area
- `volume`, total volume/area of the mesh

If several mesh cases are available in the model object, specify the mesh tag:

```
stats = mphmeshstats(model, <meshtag>);
```

Getting and Setting Mesh Data

The function `mphmeshstats` also returns the mesh data such as element coordinates. Use the function with two output variable to get the mesh data. Enter

```
[meshstats,meshdata] = mphmeshstats(model);
```

where `meshdata` is a MATLAB structure with the following fields:

- `vertex`, which contains the mesh vertex coordinates
- `elem`, which contains the element data information
- `elementity`, which contains the element entity information for each element type

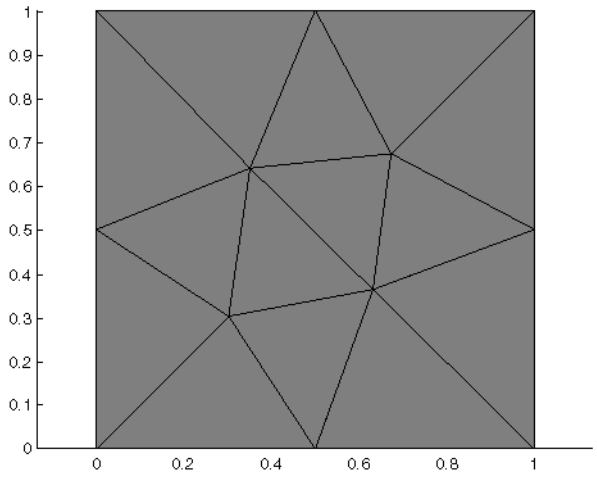
EXAMPLE: EXTRACT AND CREATE MESH INFORMATION

You can manually create a mesh based on a grid generated in MATLAB. However, before inserting this mesh into the model, a default coarse mesh is generated to get the mesh information. This information enables you to understand the requested mesh structure to use with the `createMesh` method. Then a complete mesh can be constructed and stored in the meshing sequence. If the geometry is not empty, the new mesh is checked to ensure that it matches the geometry. In summary, to create an arbitrary mesh, an empty geometry sequence and a corresponding empty meshing sequence need to be created and then the mesh is constructed on the empty meshing sequence.

Start by creating a 2D model containing a square, and mesh it with triangles:

```
model = ModelUtil.create('Model');
model.modelNode.create('mod1');
geom1 = model.geom.create('geom1', 2);
geom1.feature.create('sq1', 'Square');
geom1.run;
mesh1 = model.mesh.create('mesh1', 'geom1');
mesh1.feature.create('ftri1', 'FreeTri');
mesh1.feature.feature('size').set('hmax', '0.5');
mesh1.run('ftri1');
```

```
mphmesh(model);
```



To get the mesh data information, enter:

```
[meshstats,meshdata] = mphmeshstats(model);

meshdata =
        vertex: [2x12 double]
          elem: {[2x8 int32]  [3x14 int32]  [0 5 7 11]}
    elementity: {[8x1 int32]  [14x1 int32]  [4x1 int32]}
```

The mesh node coordinates are stored in the `vertex` field:

```
vtx = meshdata.vertex

vtx =
  Columns 1 through 7
       0    0.5000    0.3024         0    0.6314    1.0000    0.3511
       0         0    0.3023    0.5000    0.3632         0    0.6397
  Columns 8 through 12
0    0.6730    1.0000 0.5000    1.0000
1.0000    0.6728    0.5000 1.0000    1.0000
```

In the `elem` field the element information is retrieved, such as the node indices (using a 0 based) connected to the elements:

```
tri = meshdata.elem{2}
tri =
  Columns 1 through 5
          0           3           1           1           6
```

```
         1           0           4           5           3
         2           2           2           4           2
Columns 6 through 10
         6           7           6           5           9
         2           3           4           9           8
         4           6           8           4           4
Columns 11 through 14
        10          10           9          11
         7           6          11          10
         6           8           8           8
```

In the above command, observe that element number 1 is connected to nodes 1, 2, and 3, and element number 2 is connected to nodes 4, 1, and 3.

Now create manually a mesh using a data distribution generated in MATLAB. Enter the command:

```
[x,y] = meshgrid([0 0.5 1], [0 0.5 1]);
X = reshape(X,1,9);
Y = reshape(Y,1,9);
coord=[X;Y];
```

The node distribution obtained with this command corresponds to the mesh in Figure 3-18.

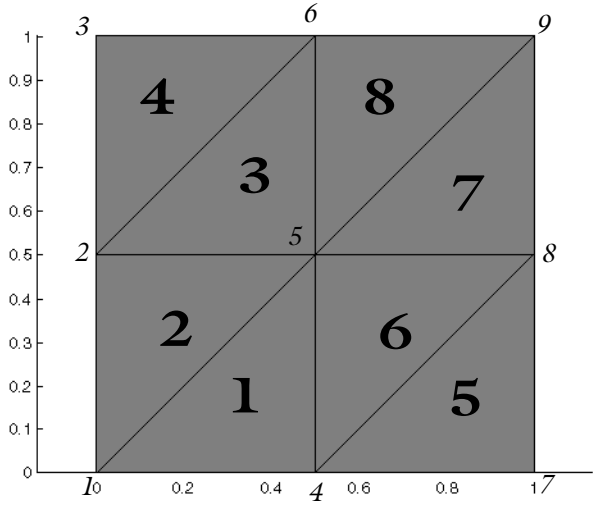


Figure 3-18: Mesh with elements (bold) and nodes (italic) indices.

Table 3-1 lists the nodes and element connectivity in the mesh.

TABLE 3-1: ELEMENT AND NODES CONNECTIVITY

ELEMENT	NODES
1	1, 4, 5
2	1, 2, 5
3	2, 5, 6
4	2, 3, 6
5	4, 7, 8
6	4, 5, 8
7	5, 8, 9
8	5, 6, 9

To create the elements and nodes connectivity information use the command:

```
new_tri(:,1)=[0;3;4];
new_tri(:,2)=[0;1;4];
new_tri(:,3)=[1;4;5];
new_tri(:,4)=[1;2;5];
new_tri(:,5)=[3;6;7];
new_tri(:,6)=[3;4;7];
new_tri(:,7)=[4;7;8];
new_tri(:,8)=[4;5;8];
```

Assign the element information, node coordinates, and elements connectivity information, into a new mesh. Use the method `createMesh` to create the new mesh.

```
geom2 = model.geom.create('geom2',2);
mesh2 = model.mesh.create('mesh2','geom2');
mesh2.data.setElem('tri',new_tri)
mesh2.data.setVertex(coord)
mesh2.data.createMesh
```

Modeling Physics

This section describes how to set up physics interfaces in a model. The physics interface defines the equations that COMSOL solves.

- The Physics Interface Syntax
- The Material Syntax
- Modifying the Equations
- Adding Global Equations
- Defining Model Settings Using External Data File

See Also

Overview of the Physics in the *COMSOL Multiphysics User's Guide*

Important

The links to features described outside of this user guide do not work in the PDF, only from within the online help.

The Physics Interface Syntax

Create a physics interface instance using the syntax

```
model.physics.create(<phystag>, physint, <geomtag>);
```

where `<phystag>` is a string that identifies the physics interface. Once defined, you can always refer to a physics interface, or any other feature, by its tag. The string `physint` is the *constructor name* of the physics interface. To get the constructor name, the best way is to create a model using the desired physics interface in the GUI and save the model as an M-file. The string `<geomtag>` refers to the geometry where you want to specify the interface.

To add a feature to a physics interface, use the syntax

```
model.physics(<phystag>).feature.create(<ftag>,operation);
```

where `the <phystag>` string refers to a physics interface. `<ftag>` is a string that you use to refer to the operation. To set a property to a value in a operation, enter:

```
model.physics(<phystag>).feature(<ftag>).set(property, <value>);
```

where <ftag> is the string that identifies the feature.

See Also

There are alternate syntaxes available. See `model.physics()` in the *COMSOL Java API Reference Guide*

To disable or remove a feature node, use the methods `active` or `remove`, respectively.

The command

```
model.physics(<phystag>).feature(<ftag>).active(false);
```

disables the feature <ftag>.

To activate the feature node you can set the `active` method to true:

```
model.physics(<phystag>).feature(<ftag>).active(true);
```

To remove a feature from the model, use the method `remove`:

```
model.physics(<phystag>).feature.remove(<ftag>);
```

EXAMPLE: IMPLEMENT AND SOLVE A HEAT TRANSFER PROBLEM

This example details how to add a physics interface and set the boundary conditions in the model object.

Start to create a model object including a 3D geometry. The geometry consists in a block with default settings. Enter the following commands at the MATLAB prompt:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1', 3);
geom1.feature.create('blk1', 'Block');
geom1.run;
```

Add a **Heat Transfer in Solids** physics interface to the model:

```
phys = model.physics.create('ht', 'HeatTransfer', 'geom1');
```

The tag of the interface is `ht`. The physics interface constructor is `HeatTransfer`. The physics is defined on geometry `geom1`.

The physics interface automatically creates a number of default features. To examine these, enter:

```
>> model.physics('ht')
```

```
ans =
Type: Heat Transfer in Solids
Tag: ht
Identifier: ht
Operation: HeatTransfer
Child nodes: solid1, ins1, cib1, init1, os1
```

The physics method has the following child nodes: `solid1`, `ins1`, `cib1`, `init1`, and `os1`. These are the default features that come with the Heat Transfer in Solids interface. The first feature, `solid1`, consists of the heat balance equation. Confirm this by entering:

```
>> solid = phys.feature('solid1')
ans =
Type: Heat Transfer in Solids
Tag: solid1
```

You can modify the settings of the `solid1` feature node, for example, to manually set the material property. To change the thermal conductivity to 400 W/(m*K) enter:

```
solid.set('k_mat', 1, 'userdef');
solid.set('k', '400');
```

The Heat Transfer in Solids interface contains features you can use to specify domain or boundary settings. For example, to add a heat source of 10^5 W/m^3 in the study domain, enter the commands:

```
hs = phys.feature.create('hs1', 'HeatSource', 3);
hs.selection.set([1]);
hs.set('Q', 1, '1e5');
```

To create a temperature boundary condition on boundaries 3, 5, and 6, enter:

```
temp = phys.feature.create('temp1', 'TemperatureBoundary', 2);
temp.selection.set([3 5 6]);
temp.set('T0', 1, '300[K]');
```

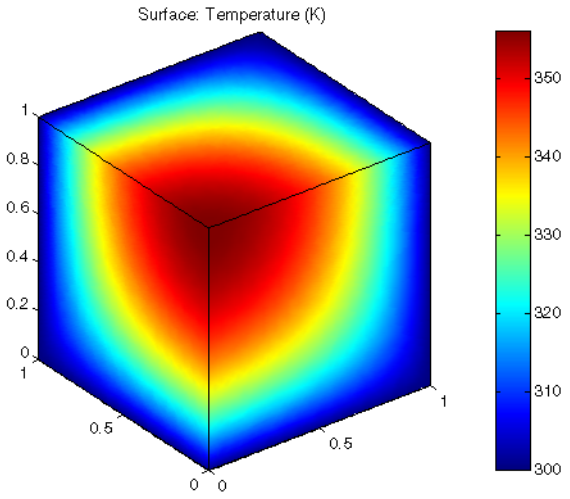
Then add a mesh and a study feature and compute the solution:

```
model.mesh.create('mesh1', 'geom1');
std = model.study.create('std1');
std.feature.create('stat', 'Stationary');
std.run
```

To visualize the solution, first create a 3D surface plot group, which is displayed in a MATLAB figure with the function `mphplot`:

```
pg = model.result.create('pg1', 'PlotGroup3D');
pg.feature.create('surf1', 'Surface');
```

```
mphplot(model,'pg1','rangenum',1)
```



The Material Syntax

In addition to changing material properties directly inside the physics interfaces, materials available in the entire model can also be created. Such a material can be used by all physics interfaces in the model.

Create a material using the syntax

```
model.material.create(<mattag>);
```

where <*mattag*> is a string that you use to refer to a material definition.

A Material is a collection of material models, where each material model defines a set of material properties, material functions, and model inputs. To add a material model, use the syntax

```
model.material(<mattag>).materialmodel.create(<mtag>);
```

where <*mattag*> is the string identifying the material defined when creating the material. The string <*mtag*> refers to the material model.

To define material properties for the model, set the property value pairs by entering:

```
model.material(<mattag>).materialmodel(<mtag>).set(property,
<value>);
```

See Also

`model.material()` in the *COMSOL Java API Reference Guide*

EXAMPLE: CREATE A MATERIAL NODE

The section Example: Implement and Solve a Heat Transfer Problem showed how to change a material property inside a physics interface. This example shows how to define a material available globally in the model. These steps assume that the previous example has been followed. Enter:

```
mat = model.material.create('mat1');
```

The material automatically creates a material model, `def`, which can be used to set up basic properties. For example, use it to define the density and the heat capacity:

```
mat.materialmodel('def').set('density', {'400'});
mat.materialmodel('def').set('heatcapacity', {'2e3'});
```

To use the defined material in a model, set the `solid1` feature to use the material node. Enter:

```
solid.set('k_mat',1,'from_mat');
```

Modifying the Equations

The equation defining the physics node can be edited with the method `featureInfo('info')` applied to a feature of the physics node `physics(<phystag>).feature(<ftag>)`, where `<phystag>` and `<ftag>` identify the physics interface and the feature, respectively.

```
info =
model.physics(<phystag>).feature(<ftag>).featureInfo('info');
```

Use the method `getInfoTable(type)` to return the tables available in the **Equation View** node

```
infoTable = info.getInfoTable(type);
```

where *`type`* defines the type of table to return. It can have the value `'Weak'` to return the weak form equations, `'Constraint'` to return the constraint types table, or `'Expression'` to return the variable expressions table.

EXAMPLE: ACCESS AND MODIFY THE EQUATION WEAK FORM

This example continues from the Example: Implement and Solve a Heat Transfer Problem and modifies the model equation.

To retrieve information about the physics interface create an info object:

```
info = model.physics('ht').feature('solid1').featureInfo('info');
```

From the `info` object access the weak form equation by entering:

```
infoTable = info.getInfoTable('Weak');
```

This returns a string variable that contains both the name of the weak equation variable and the equation of the physics implemented in the weak form. Enter the command

```
list = infoTable(:)
```

which results in the output:

```
list =
java.lang.String[]:
    [1x159 char]
    'root.mod1.ht.solid1.weak$1'
    'Material'
    'Domain 1'
```

The output shows that the physics is defined with the weak expression available in the variable list(1). Enter

```
list(1)
```

to get the weak equation as a string variable. The result of this command is:

```
ans =
-(ht.k_effxx*Tx+ht.k_effxy*Ty+ht.k_effxz*Tz)*test(Tx)-(ht.k_effyx
*Tx+ht.k_effyy*Ty+ht.k_effyz*Tz)*test(Ty)-(ht.k_effzx*Tx+ht.k_eff
zy*Ty+ht.k_effzz*Tz)*test(Tz)
```

To access the equation in the node `root.mod1.ht.solid1.weak$1`; for example, to modify the equation and lock the expression, run the commands:

```
equExpr = '400[W/(m*K)]*(-Tx*test(Tx)-Ty*test(Ty)-Tz*test(Tz))';
info.lock(list(2), {equExpr});
```

These commands set the heat conductivity to a constant value directly within the heat balance equation.

Adding Global Equations

To add a global equation in the model use the command:

```
model.physics.create(<odestag>, 'GlobalEquations');
```

To define the name of the variable to be solved by the global equation, enter

```
model.physics(<odetag>).set('name', <idx>, <name>);
```

where <idx> is the index of the global equation, and <name> a string with the name of the variable.

Set the expression <expr> of the global equation with

```
model.physics(<odetag>).set('equation', <idx>, <expr>);
```

where <expr> is defined as a string variable.

Initial value and initial velocity can be set with the commands

```
model.physics(<odetag>).set('initialValueU', <idx>, <init>);
model.physics(<odetag>).set('initialValueUt', <idx>, <init_t>);
```

where <init> and <init_t> are the initial value expression for the variable and its time derivative respectively.

EXAMPLE: SOLVE AN ODE PROBLEM

This example illustrates how to solve the following ODE in a COMSOL model:

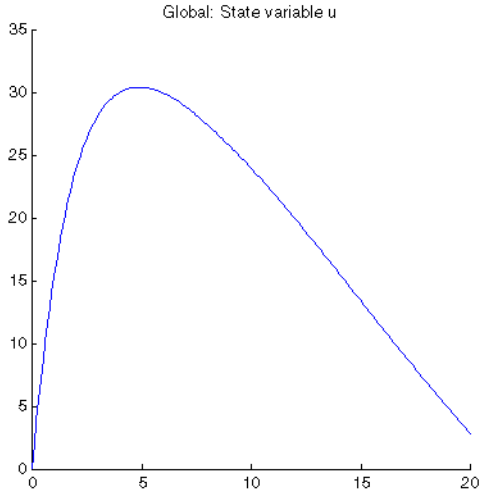
$$\ddot{u} + \frac{\dot{u}}{2} + 1 = 0$$

$$u0 = 0$$

$$\dot{u}0 = 20$$

```
model = ModelUtil.create('Model');

ge = model.physics.create('ge', 'GlobalEquations');
ge1 = ge.feature('ge1');
ge1.set('name', 1, 1, 'u');
ge1.set('equation', 1, 1, 'utt+0.5*ut+1');
ge1.set('initialValueU', 1, 1, 'u0');
ge1.set('initialValueUt', 1, 1, 'u0t');

model.param.set('u0', '0');
model.param.set('u0t', '20');

std1 = model.study.create('std1');
std1.feature.create('time', 'Transient');
std1.feature('time').set('tlist', 'range(0,0.1,20)');
std1.run;
```

```
model.result.create('pg1', 1);
model.result('pg1').set('data', 'dset1');
model.result('pg1').feature.create('glob1', 'Global');
model.result('pg1').feature('glob1').set('expr', {'mod1.u'});

mphplot(model,'pg1')
```



Defining Model Settings Using External Data File

To use tabulated data from files in a model, use the interpolation function available under the Global Definitions node or the Definitions node of the model.

To add an interpolation function to the manual, enter:

```
model.func.create(<functag>, 'Interpolation');
```

The interpolation function is initially defined globally and is located in the Model Builder on the COMSOL Desktop under the Global Definitions node. If you have several model nodes in your model and you want to attach it to the specified model node `<model>`, enter:

```
model.func(<functag>).model(<model>);
```

where `<model>` is the tag of the model node to attach the interpolation function.

Then you can interpolate data specified by a table inside the function (default), or specified in an external file.

When using an interpolation table, set the interpolation data for each row of the table with the commands:

```
model.func(<functag>).setIndex('table', <t_value>, <i>, 1);
model.func(<functag>).setIndex('table', <ft_value>, <i>, 2);
```

where *<t_value>* is the interpolation parameter value and *<ft_value>* is the function value. *<i>* is the index (0-based) in the interpolation table.

To use an external file change the source for the interpolation and specify the file, where *filename* is the name (including the path) of the data file:

```
model.func(<functag>).set('source', 'file');
model.func(<functag>).set('filename', <filename>);
```

Several interpolation methods are available. Choose the one to use with the command:

```
model.func(<functag>).set('interp', method);
```

The string *method* can be set as one of the following alternatives:

- `'neighbor'`, for interpolation according to the nearest neighbor method
- `'linear'`, for linear interpolation method
- `'cubicspline'`, for cubic spline interpolation method
- `'piecewisecubic'`, piecewise cubic interpolation method

You can also decide how to handle parameter values outside the range of the input data by selecting an extrapolation method:

```
model.func(<functag>).set('extrap', method);
```

The string *method* can be one of these values:

- `'const'`, to use a constant value outside the interpolation data
- `'linear'`, for linear extrapolation method
- `'nearestfunction'`, to use the nearest function as extrapolation method
- `'value'`, to use a specific value outside the interpolation data

See Also

`model.func()` in the *COMSOL Java API Reference Guide*

Creating Selections

In this section:

- The Selection Node
- Coordinate-Based Selections
- Selection Using Adjacent Geometry
- Displaying Selections

See Also

User-Defined Selections in the *COMSOL Multiphysics User's Guide*

Important

The links to features described outside of this user guide do not work in the PDF, only from within the online help.

The Selection Node

Use a **Selection** node to define a collection of geometry entities in a central location in the model. The selection can easily be accessed in physics or mesh features or during results analysis. For example, you can refer collectively to a set of boundaries that have the same boundary conditions, which also have the same mesh size settings.

A selection feature can be one of these types:

- explicit, to include entities explicitly defined by their definitions indices
- ball, to include entities that fall with a set sphere
- box, to include entities that fall within a set box

Selection can also be combined by Boolean operations, such as **Union**, **Intersection**, and **Difference**.

SETTING AN EXPLICIT SELECTION

Create an explicit selection with the command:

```
model.selection.create(<seltag>, 'Explicit');
```

To specify the domain entity dimension to use in the selection node, enter

```
model.selection(<seltag>).geom(sdim);
```

where *sdim* is the space dimension that represents the different geometric entities:

- 3 for domains
- 2 for boundaries/domains
- 1 for edges/boundaries, and
- 0 for points.

Set the domain entity indices in the selection node with the command

```
model.selection(<seltag>).set(<idx>);
```

where <idx> is an array of integers that list the geometric entity indices to add in the selection.

Coordinate-Based Selections

DEFINING A BALL SELECTION NODE

The **Ball** selection node is defined by a center point and a radius. The selection can include geometric entities that are completely or partially inside the ball. The selection can be set up by using either the COMSOL API directly or the `mphselectcoords` function.

Ball Selection Using the COMSOL API

To add a ball selection to a model object enter:

```
model.selection.create(<seltag>, 'Ball');
```

To set the coordinates (<x0>, <y0>, <z0>) of the selection center point, enter

```
model.selection(<seltag>).set('posx', <x0>);
model.selection(<seltag>).set('posy', <y0>);
model.selection(<seltag>).set('posz', <z0>);
```

where <x0>, <y0>, <z0> are double values.

Specify the ball radius <r0> with the command

```
model.selection(<seltag>).set('r', <r0>);
```

where <r0> is a double floating-point value.

To specify the geometric entity level, enter

```
model.selection(<seltag>).set('entitydim', edim);
```

where *edim* is an integer defining the space dimension value (3 for domains, 2 for boundaries/domains, 1 for edges/boundaries, and 0 for points).

The selection also specifies the condition for geometric entities to be selected:

```
model.selection(<seltag>).set('condition', condition);
```

where *condition* can be:

- 'inside', to select all geometric entities completely inside the ball,
- 'intersects', to select all geometric entities that intersect the ball (default),
- 'somevertex', to select all geometric entities where at least some vertex is inside the ball, or
- 'allvertices', to select all geometric entities where all vertices are inside the ball.

Ball Selection Using MPHSELECTCOORDS

The function mphselectcoords retrieves geometric entities enclosed by a ball.

To get the geometric entities enclosed by a ball of radius r0, with its center positioned at (x0,y0,z0) enter the command

```
idx = mphselectcoords(model, <geomtag>, [<x0>,<y0>,<z0>], ...
entitytype,'radius',<r0>);
```

where <*geomtag*> is the tag of geometry where the selection, and *entitytype*, can be one of 'point', 'edge', 'boundary', or 'domain'.

The above function returns the entity indices list. Use it to specify a feature selection or to create an explicit selection as described in Setting an Explicit Selection.

By default the function searches for the geometric entity vertices near these coordinates using the tolerance radius. It returns only the geometric entities that have all vertices inside the search ball. To include any geometric entities in the selection that have at least one vertex inside the search ball, set the property include to 'any':

```
idx = mphselectcoords(model, <geomtag>, [<x0>,<y0>,<z0>], ...
entitytype,'radius',<r0>,'include','any');
```

In case the model geometry is finalized as an assembly, you have distinct geometric entities for each part of the assembly (pair). Specify the adjacent domain index to avoid selection of any overlapping geometric entities. Set the adjnumber property with the domain index:

```
idx = mphselectcoords(model, <geomtag>, [<x0>,<y0>,<z0>], ...
entitytype,'radius',<r0>,'adjnumber',<idx>);
```

where <*idx*> is the domain index adjacent to the desired geometric entities.

DEFINING A BOX SELECTION NODE

The **Box** selection node is defined by two diagonally opposite points of a box (in 3D) or rectangle (in 2D).

Box Selection Using the COMSOL API

This command adds a box selection to the model object:

```
model.selection.create(<seltag>, 'Box');
```

To specify the points (<*x0*>, <*y0*>, <*z0*>) and (<*x1*>, <*y1*>, <*z1*>), enter

```
model.selection(<seltag>).set('xmin', <x0>);
model.selection(<seltag>).set('ymin', <y0>);
model.selection(<seltag>).set('zmin', <z0>);
model.selection(<seltag>).set('xmax', <x1>);
model.selection(<seltag>).set('ymax', <y1>);
model.selection(<seltag>).set('zmax', <z1>);
```

where <*x0*>, <*y0*>, <*z0*>, <*x1*>, <*y1*>, <*z1*> are double values.

To specify the geometric entities levels use the command

```
model.selection(<seltag>).set('entitydim', edim);
```

where *edim* is an integer defining the space dimension value (`3` for domains, `2` for boundaries/domains, `1` for edges/boundaries, and `0` for points).

The selection also specifies the condition for geometric entities to be selected:

```
model.selection(<seltag>).set('condition', condition);
```

where *condition* can be:

- `'inside'`, to select all geometric entities completely inside the ball,
- `'intersects'`, to select all geometric entities that intersect the ball (default),
- `'somevertex'`, to select all geometric entities where at least some vertex is inside the ball, or
- `'allvertices'`, to select all geometric entities where all vertices are inside the ball.

Box Selection Using MPHSELECTBOX

The function `mphselectbox` retrieves geometric entities enclosed by a box (in 3D) or rectangle (in 2D).

To get the geometric entities of type *entitytype* enclosed by the box defined by the points (`x0,y0,z0`) and (`x1,y1,z1`), enter the command:

```
idx = mphselectbox(model,<geomtag>,[<x0> <x1>,<y0> <y1>,<z0> <z1>],
entitytype);
```

where <*geomtag*> is the geometry tag where the selection is applied, and *entitytype* can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'`.

The above function returns the entity indices list. Use it to specify a feature selection or to create an explicit selection as described in Setting an Explicit Selection.

By default the function searches for the geometric entity vertices near these coordinates using the tolerance radius. It returns only the geometric entities that have all vertices inside the box or rectangle. To include any geometric entities in the selection that have at least one vertex inside the search ball, set the property `include` to `'any'`:

```
idx = mphselectbox(model,<geomtag>,[<x0> <x1>,<y0> <y1>,<z0> <z1>],
entitytype,'include','any');
```

In case the model geometry is finalized as an assembly (pair), you have distinct geometric entities for each part of the assembly. Specify the adjacent domain index to avoid selection of overlapping geometric entities. Set the `adjnumber` property with the domain index

```
idx = mphselectbox(model,<geomtag>,[<x0> <x1>,<y0> <y1>,<z0> <z1>],
entitytype, 'adjnumber', <idx>);
```

where <*idx*> is the domain index adjacent to the desired geometric entities.

Selection Using Adjacent Geometry

Another approach to select geometric entities is to define the adjacent object. For example, select edges adjacent to a specific domain, or boundaries that are adjacent to a specific point.

Adjacent Selection Using the COMSOL API

This command creates a selection node using adjacent geometric entities:

```
model.selection.create(<seltag>, 'Adjacent');
```

The geometric entity level needs to be specified with the command

```
model.selection(<seltag>).set(edim);
```

where *edim* is an integer defining the space dimension value (`3` for domains, `2` for boundaries/domains, `1` for edges/boundaries, and `0` for points).

The **Adjacent** selection node only supports the Selection node as an input:

```
model.selection(<seltag>).set( 'Adjacent');
```

and specify the ball radius <*r0*> with the command

```
model.selection(<seltag>).set('input', <seltag>);
```

where <*seltag*> is the tag of an existing Selection node.

Select the level of geometric entities to add in the selection with the command

```
model.selection(<seltag>).set('outputdim', edim);
```

where *edim* is an integer defining the space dimension value (`3` for domains, `2` for boundaries/domains, `1` for edges/boundaries, and `0` for points).

If there are multiple domains in the geometry to include in the interior and exterior selected geometric entities, then enter:

```
model.selection(<seltag>).set('interior', 'on');
model.selection(<seltag>).set('exterior', 'on');
```

To exclude the interior/exterior, select geometric entities and set the respective property to `'off'`.

Adjacent Selection Using MPHGETADJ

An alternative to the COMSOL API is to use the function `mphgetadj` to select geometric entities using an adjacent domain.

To get a list of entities of type *entitytype* adjacent to the entity with the index <*adjnumber*> of type *adjtype*, enter

```
idx = mphselectbox(model, <geomtag>, entitytype, ...
adjtype, <adjnumber>);
```

where <*geomtag*> is the tag of geometry where the selection applies. The string variables *entitytype* and *adjtype* can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'`.

The list returned by the function can be used to specify the selection for a model feature or to create an explicit selection as described in Setting an Explicit Selection.

Displaying Selections

Use the function `mphviewselection` to display the selected geometric entities in a MATLAB figure.

You can either specify the geometry entity index and its entity type or specify the tag of a selection node available in the model.

To display the entity of type *entitytype* with the index <*idx*> enter

```
mphviewselection(model, <geomtag>, <idx>, 'entity', entitytype)
```

where <*geomtag*> is the geometry node tag, and <*idx*> is a positive integer array that contains the entity indices. The string *entitytype* can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'`.

If the model contains a selection node with the tag <*seltag*>, it can be displayed with the command:

```
mphviewselection(model, <geomtag>, <seltag>)
```

If the selection node is a Ball or Box selection, the ball or box used in the selection can be displayed with this command:

```
mphviewselection(model, <geomtag>, <seltag>, 'showselector', 'on')
```

Computing the Solution

This section describes the commands to use to compute the solution at the MATLAB prompt. How to set up and run a study node but also how to set manual solver sequence. This includes the following paragraphs:

- The Study Node
- The Solver Sequence Syntax
- Run the Solver Sequence
- Adding a Parametric Sweep
- Adding a Job Sequence
- Plot While Solving

See Also

- Solver Features in the *COMSOL Multiphysics Reference Guide*
- Solver in the *COMSOL Java API Reference Guide*

Important

The links to features described outside of this user guide do not work in the PDF, only from within the online help.

The Study Node

A study node holds the nodes that define how to solve a model. These nodes are divided into three broad categories:

- Study steps, which determines overall settings suitable for a certain study type.
- Solver sequence.
- Job configurations for distributed parametric jobs, batch jobs, and cluster computing.

See Also

- The Study Node in the *COMSOL Reference Guide*

Create a study node by using the syntax

```
model.study.create(<studytag>);
```

where *studytag* is a string that is used to define the study node.

The minimal definition for the study node consists in a study step that define the type of study to use to compute the solution. To add a study step to the study node, use the syntax

```
model.study(<studytag>).feature.create(<ftag>, operation);
```

where *<studytag>* is the string identifying the study node. The string *<ftag>* is a string that is defined to refer to the study step. The string *operation* is one of the basic study types, such as `Stationary`, `Transient` or `Eigenfrequency`, and more.

See Also

Study Types in the *COMSOL Reference Guide*

To specify a property value pair for a study step, enter

```
model.study(<studytag>).feature(<ftag>).set(property, <value>);
```

where *<ftag>* is the string identifying the study step.

To generates the default solver sequence associated with the physics solved in the model and compute the solution, run the study node with the command

```
model.study(<studytag>).run
```

See Also

model.study() in the *COMSOL Java API Reference Guide*

The Solver Sequence Syntax

If you don't want to use the default solver sequence created by the study node, you have the possibility to manually create one. To create a solver sequence, enter

```
model.sol.create(<soltag>);
```

where `<soltag>` is a string used to refer to the solver sequence associated to a solution object.

A solver sequence has to be connected to a study node, this can be done with the command

```
model.sol(<soltag>).study(<studytag>);
```

where `<studytag>` is the tag of the study you want to associate the solver sequence defined with the tag `<soltag>`.

A solver sequence also requires the definition of the following nodes:

- Study Step, where you specify which study and study step to use when compiling the equations for and computing the current solver sequence.
- Dependent Variables, this node handles settings for the computation of dependent variables, such as initial value and scaling settings but also the dependent variables not solved for.
- Solver node, where you specify the type of solver to use to compute the solution.

Add the nodes to the solver sequence with the command

```
model.sol(<soltag>).feature.create(<ftag>, operation);
```

where `<soltag>` is the string defined when creating the solver sequence. The string *`ftag`* is a string that is defined to refer to the node, for example, a study step. *`operation`* can be one of the following `'StudyStep'`, `'Variables'` or `'Stationary'`.

To specify a property value pair for a solver feature, enter

```
model.sol(<soltag>).feature(<ftag>).set(property, <value>);
```

where <soltag> is a string referring to the solver sequence configuration.

See Also

For a list of the operations available for the solver node, see Features Producing and Manipulating Solutions and Solver, in the *COMSOL Java API Reference Guide*.

Run the Solver Sequence

There are several ways to run the solver sequence: run the entire sequence, run up to a specified feature or run from a specified feature.

Use the methods `run` or `runAll` to run the entire solver configuration node:

```
model.sol(<soltag>).run;
model.sol(<soltag>).runAll;
```

You can also use the method `run(<ftag>)` to run the solver sequence up to the solver feature with the tag <ftag>:

```
model.sol(<soltag>).run(<ftag>);
```

For those occasions when you want to continue solving a sequence, use the method `runFrom(<ftag>)` to run the solver configuration from the solver feature with the tag <ftag>:

```
model.sol(<soltag>).runFrom(<ftag>)
```

Adding a Parametric Sweep

In addition to the study step that defines a study type, you can add to the study node a parametric sweep. This is a study step that does not generate equations and can only be used in combination with other study steps. You can formulate the sequence of problems that arise when some parameters are varied in the model.

To add a parametric sweep to the study node, enter:

```
model.study(<studytag>).feature.create(<ftag>, 'Parametric');
```

where <studytag> is the tag of the study node where to include the parametric sweep defined with the tag <ftag>.

To add one or several parameters to the sweep, enter the command

```
model.study(<studytag>).feature(<ftag>).setIndex('pname',
<pname>, <idx>);
```

where <pname> is the name of the parameter to use in the parametric sweep and <idx> the index number of the parameter. Set the <idx> to 0 to define the first parameter, 1 to define the second parameter, and so on.

Set the list of the parameter values with the command

```
model.study(<studytag>).feature(<ftag>).setIndex('plistarr',
<pvalue>, <idx>);
```

where <pvalue> contains the list of parameter values defined with either a string or with a double array, and <idx> is the index number of the parameter and uses the same value as for the parameter name.

If there are several parameters listed in the parametric sweep node, select the type of sweep by entering:

```
model.study(<studytag>).feature(<ftag>).set('sweeptype', type);
```

where the sweep type,*type*, can be either 'filled' or 'sparse', referring to all combination or specified combinations of the parameter values respectively.

Adding a Job Sequence

In the study node you can define a job sequence such as distributed parametric jobs, batch jobs and cluster computing. To create a batch node enter

```
model.batch.create(<batchtag>, type);
```

where <batchtag> is the tag of the job sequence and *type* is the type of job to define. It can be either Parametric, Batch, or Cluster.

As for a solver sequence you need to attach the job sequence to an existing study node, this is done with the command

```
model.batch(<batchtag>).atached(<studytag>);
```

where <studytag> is the tag of the study node.

For each job type, such as parametric, batch or cluster job can be defined with specific properties. Use the Set method to add a property to the batch job:

```
model.batch(<batchtag>).set(property, <value>);
```

You can get the list of the properties in model.batch() in the *COMSOL Java API Reference Guide*.

To run the batch sequence use the run method:

```
model.batch(<batchtag>).run;
```

Plot While Solving

With the Plot While Solving functionality you can monitor the development of the computation by updating predefined plots during the computation. Since the plots are displayed on a COMSOL graphics window, start COMSOL with MATLAB using the graphics mode.

See the *COMSOL Multiphysics Installation and Operations Guide* to start COMSOL with MATLAB with the graphics mode.

To activate Plot While Solving, enter the command

```
model.study(<studytag>).feature(<studysteptag>).set('plot',
'on');
```

where <*studytag*> and <*studysteptag*> refer to the study node and study step, respectively.

Specify the plot group to plot by setting the plot group tag:

```
model.study(<studytag>).feature(<studysteptag>).set('plotgroup',
<ptag>);
```

Only one plot group can be plotted during a computation. Use the probe feature instead if you need to monitor several variables.

To activate Plot While Solving for a probe plot, enter the command

```
model.study(<studytag>).feature(<studysteptag>).set('probesel',
seltype);
```

where *seltype* is the type of probe selection, that can be `'none'`, `'all'` or `'manual'`.

In case the probe selection is set to manual you have to specify the list of the probe variable to display. This is done with the command

```
model.study(<studytag>).feature(<studysteptag>).set('probes',
<list>);
```

where <*list*> is the a cell array containing the list of the probe to use.

Analyzing the Results

This section describes how to do results analysis and visualization.

- The Plot Group Syntax
- Displaying The Results
- The Data Set Syntax
- The Numerical Node Syntax
- Exporting Data

See Also

- Results Evaluation and Visualization in the *COMSOL Multiphysics User's Guide*
- Results in the *COMSOL Java API Reference Guide*

Important

The links to features described outside of this user guide do not work in the PDF, only from within the online help.

The Plot Group Syntax

Result plots always appear in plot groups, which you add to the model by the create method:

```
model.result.create(<pgtag>, sdim);
```

select the string `<pgtag>` to identify the plot group, and the integer `sdim` to set the space dimension (1, 2, or 3) of the group.

To add a plot to a plot group, use the syntax

```
model.result(<pgtag>).feature.create(<ftag>, plottype);
```

where the string `<ftag>` identifies the plot, and the string `plottype` defines its type.

Plots can have different attributes that modify the display. For instance the Deformation attribute deforms the plot according to an evctor quantity, the Height Expression attribute introduces 3D height on a 2D table surface plot and the Filter

attribute filters the plot using a logical expression. The type of plot determines which attributes are available. Add an attribute to a plot by the command:

```
model.result(<pgtag>).feature(<ftag>).feature.create(<attrtag>,
attrtype);
```

where *attrtype* is a string that defines the attribute type.

See Also

For a list of available plot types and corresponding attribute types, see Results in the *COMSOL Java API Reference Guide*. Also see model.result() in this guide.

Displaying The Results

DISPLAY PLOT GROUPS

Use the command `mphplot` to display a plot group available from model object. For example to display the plot group <*pgtag*> enter:

```
mphplot(model, <pgtag>);
```

This renders the graphics in a MATLAB figure window. In addition you can plot results in a COMSOL Graphics window, if you start COMSOL with MATLAB in graphics mode. To do this for a plot group <*pgtag*> enter:

```
mphplot(model, <pgtag>, 'server', 'on');
```

See Also

See the *COMSOL Multiphysics Operations and Installation Guide* to start COMSOL with MATLAB in graphics mode.

Another way to plot in a COMSOL Graphics Window is to use the run method:

```
model.result(<pgtag>).run;
```

Note

Plotting in a COMSOL Graphics window is not supported on Mac OS.

The default settings for plotting in a MATLAB figure do not display the color legend. To include the color legend in a figure, use the property `rangenum`

```
mphplot(model, <pgtag>, 'rangenum', <idx>);
```

where the integer `<idx>` identifies the plot for which the color legend should be displayed.

EXTRACT PLOT DATA

In some situation it can be useful to extract data from a plot, for example if you to manually edit the plot as it is allowed in MATLAB. To get a cell array, `dat`, which contains the data for each plot feature available in the plot group `<pgtag>` type

```
dat = mphplot(model, <pgtag>);
```

PLOT EXTERNAL DATA

Using the function `mphplot` you can also plot data that you specify directly as an input argument. The supported data format is according to the structure provided by the function `mpheval`. This allows you to plot data that you have first extracted from the model then modified in MATLAB, on the model geometry. To plot the structure `<data>`, run the command:

```
mphplot(<data>);
```

If the data structure contains the value of several expressions, set the one to display in the plot with the index property

```
mphplot(<data>, 'index', <idx>);
```

where `<idx>` is a positive integer that corresponds to the expression to plot.

You may select from several available color tables when visualizing data, by using the colortable option:

```
mphplot(<data>, 'colortable', colorname);
```

Obtain a list of alternatives for `colorname` from the on-line help by typing

```
help colortable
```

Note

`mphplot` supports only plotting of data structures that are of the type point, line or surface evaluations from `mpheval`.

EXAMPLE: PLOT MPHEVAL DATA

This example extracts COMSOL data at the MATLAB prompt, modifies it and plots the data in a MATLAB figure.

First load the Busbar model from the COMSOL Model Library. Enter:

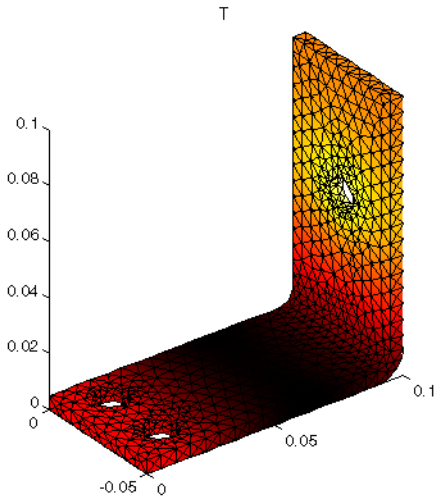
```
model = mphload('busbar');
```

To extract the temperature and the electric potential field, use the command `mpheval`:

```
dat = mpheval(model,{'T','V'},'selection',1);
```

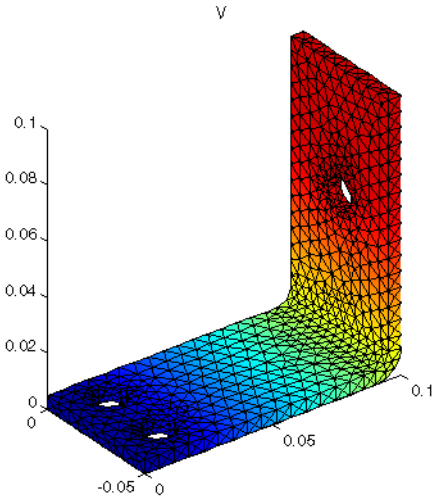
To display the temperature field, using the thermal color table:

```
mphplot(dat,'index',1,'colortable','thermal');
```



Now do a simple scaling of the electric potential then plot it using the default color table:

```
dat.d2 = dat.d2*1e-3;
mphplot(dat, 'index', 2);
```



The Data Set Syntax

Use Data Sets to make solutions and meshes available for visualization and data analysis. You can create Solution Data Set, Mesh Data Set or Visualization Data Set (such as, for instance, Cut Plane or Edge Data Sets). While Solution and Mesh Data Sets are self defined, Visualization Data Sets always refer to an existing Solution Data Set.

See Also	See Data Sets in the section Commands Grouped by Function of the *COMSOL Multiphysics Java API Reference Guide* to get a list of the available Data Sets.

Tip	All plots refer to data sets; the solutions are always available as the default data set.

To create a data set at the MATLAB prompt, use the command:

```
model.result.dataset.create(<dsettag>, dsettype);
```

where *dsettype* is one of the available data set types.

See Also

- Defining Data Sets in the *COMSOL Multiphysics User's Guide*
- Use of Data Sets in the *COMSOL Java API Reference Guide*

The Numerical Node Syntax

Use the numerical node to perform numerical evaluation from within the COMSOL model. Numerical operations such as computing averages, integrations, maximums, or minimums of a given expression are available. You can also perform point and global evaluations.

To create a numerical node, enter

```
model.result.numerical.create(<numtag>, numtype);
```

where *numtype* is the type of operation to be performed by the node.

See Also

For a list of the syntax of the numerical results type available, see About Results Commands in the *COMSOL Java API Reference Guide*.

To store the data you need to create a table and associate the table to the numerical node:

```
model.result.table.create(<tabletag>,'Table');
model.result.numerical(<numtag>).set('table',<tabletag>);
```

where <*tabletag*> is the tag of the table where you want to store the data evaluated with the numerical operations defined with the tag <*numtag*>.

If you want to extract in MATLAB the data stored in a table you can use the methods `getRealRow` and `getImagRow`, such as:

```
realRow = model.result.table(<tabletag>).getRealRow(<idx>);
imagRow = model.result.table(<tabletag>).getRealRow(<idx>);
```

where <*idx*> is the column index of the table <*tabletag*>.

For data evaluation in MATLAB you can also use the function `mpheval`, `mphevalpoint`, `mphglobal`, `mphint2`, `mphinterp`, `mphmax`, `mphmean` and `mphmin`.

See Also

Extracting Results in the chapter *Working with Models*.

Exporting Data

Use the `export` node to generate an animation or to export data to an external file (ASCII format).

ANIMATION EXPORT

Animations can be defined as two different types: a movie or an image sequence. The movie generates file formats such as GIF (.gif), AVI (.avi), or flash (.swf); the image sequence generates a sequence of images. Make sure COMSOL with MATLAB is started with the graphics mode to enable plot on server.

See Also

To learn how to start COMSOL with MATLAB with the graphics mode, see the *COMSOL Multiphysics Installation and Operations Guide*.

To generate an animation, add an `Animation` node to the `export` method:

```
model.result.export.create(<animtag>, 'Animation');
```

To change the animation type use the `'type'` property according to

```
model.result.export(<animtag>).set('type', type);
```

where *`type`* is either `'imageseq'` or `'movie'`.

To set the filename and finally create the animation, enter

```
model.result.export(<animtag>).set(typefilename, <filenname>);
model.result.export(<animtag>).run;
```

In the above *`typefilename`* depends on the type of animation export: `'imagefilename'` for an image sequence, `'giffilename'` for a gif animation, `'flashfilename'` for a flash animation, and `'avifilename'` for an avi animation.

For a `movie` type animation, it is possible to change the number of frames per second with the command

```
model.result.export(<animtag>).set('fps', <fps_number>);
```

where `<fps_number>` is a positive integer that corresponds to the number of frames per second to use.

For all animation types you can modify the width and the height of the plot with the set method:

```
model.result.export(<animtag>).set('width', <width_px>);
model.result.export(<animtag>).set('height', <height_px>);
```

where, the positive integers `<width_px>` and `<height_px>` are the width and height size (in pixels), respectively, to use for the animation.

DATA EXPORT

In order to save data to an ASCII file, create a `Data` node to the `export` method:

```
model.result.export.create(<datatag>, 'Data');
```

Set the expression *`expr`* and the file name *`filename`*, and run the export:

```
model.result.export.(<datatag>).setIndex('expr', <expr>, 0);
model.result.export.(<datatag>).set('filename', <filename>);
```

Set the export data format with the `struct` property:

```
model.result.export.(<datatag>).set('struct', datastruct);
```

where *`datastruct`* can be set to `'spreadsheet'` or `'sectionwise'`.

See Also

See Data Formats in the *COMSOL Multiphysics Reference Guide* for details about the data formats used in the exported data files.

To export the data in the specified file, run the export node:

```
model.result.export.(<datatag>).run;
```

ANIMATION PLAYER

For transient and parametric studies, an animation player can be generated to create interactive animations.

The player displays the figure on a COMSOL Graphics window. Make sure COMSOL with MATLAB is started with the graphics mode enabled.

See Also

To learn how to start COMSOL with MATLAB with the graphics mode, see the *COMSOL Multiphysics Installation and Operations Guide*.

To create a player feature node to the model enter the command:

```
model.result.export.create(<playtag>, 'Player');
```

Then associate the player with an existing plot group by setting the plot group property:

```
model.result.export(<playtag>).set('plotgroup', <pgtag>);
```

where `<pgtag>` refers to the plot group, which is animated in the player.

The default frame number used to generate the animation is 25, you can also specify the number of frame with the command

```
model.result.export(<playtag>).set('maxframe', <maxnum>);
```

where `<maxnum>` is a positive integer value that corresponds to the maximum number of frames to generate with the player.

Use the run method to generate the player:

```
model.result.export(<playtag>).run;
```

4

Working With Models

This section introduces you to the functionality available for LiveLink *for* MATLAB including the wrapper functions and the MATLAB tools that can be used and combined with a COMSOL model object.

In this chapter:

- Using MATLAB Variables in Model Settings
- Extracting Results
- Running Models in a Loop
- Running Models in Batch Mode
- Working with Matrices
- Extracting Solution Information and Solution Vector
- Retrieving Xmesh Information
- Navigating the Model
- Handling Errors and Warnings
- Improving Performance for Large Models
- Creating a Custom GUI
- COMSOL 3.5a Compatibility

Using MATLAB Variables in Model Settings

LiveLink *for* MATLAB allows you to define the model properties with MATLAB variables or a MATLAB M-function.

In this section:

- The Set and SetIndex Methods
- Using a MATLAB Function to Define Model Properties

The Set and SetIndex Methods

You can use MATLAB variables to set properties of a COMSOL model. Use the `set` or `setIndex` methods to pass the variable value from MATLAB to the COMSOL model.

THE SET METHODS

Use the `set` method to assign parameter and/or property values. All assignments return the parameter object, which means that assignment methods can be appended to each other.

The basic method for assignment is:

```
something.set(name, <value>);
```

The *name* argument is a string with the name of the parameter/property. The *<value>* argument can for example be a MATLAB integer or double array variable.

When using a MATLAB variable, make sure that the value corresponds to the model unit system. COMSOL can also take care of the unit conversation automatically; in this case convert the MATLAB integer/double variable to a string variable and use the set method as

```
something.set(property, [num2str(<value>)'[unit]']);
```

where is the *unit* you want to set the value property.

THE SETINDEX METHODS

Use the `setIndex` methods to assign values to specific indices (0-based) in an array or matrix property. All assignment methods return the parameter object, which means that assignment methods can be appended to each other:

```
something.setIndex(property, <value>, <index>);
```

The *name* argument is a string with the name of the property, `<value>` is the value to set the property, which can be a MATLAB variable value, and `<index>` is the index in the property table.

When using a MATLAB variable make sure that the value corresponds to the model unit system. COMSOL can automatically take care of the unit conversation; in this case converting the MATLAB integer/double variable to a string variable and using the set method as

```
something.setIndex(property, [num2str(<value>)'[unit]'], <index>);
```

where `[unit]` is the *unit* you want to set the value property.

Using a MATLAB Function to Define Model Properties

Use MATLAB Function to define the model property. The function can either be declared within the model object or called at the MATLAB prompt.

CALLING MATLAB FUNCTIONS WITHIN THE COMSOL MODEL OBJECT

LiveLink *for* MATLAB enables you to declare a MATLAB M-function directly from within the COMSOL model object. This is typically the case if you want to call MATLAB M-function from the COMSOL Desktop. The function being declared within the model object accepts any parameter, variable, or expression arguments defined in the COMSOL model object. However, to use a variable defined at the MATLAB prompt, the variable has to be transferred first in the COMSOL model, as a parameter for instance (see how to set a MATLAB variable in the COMSOL model in The Set and SetIndex Methods).

The function is evaluated any time the model needs to be updated. The model object cannot be called as an input argument of the M-function.

See Also

Calling MATLAB Functions

CALLING MATLAB FUNCTIONS AT THE MATLAB PROMPT

Use a MATLAB function to define a model property with the `set` method

```
something.set(property, myfun(<arg>));
```

where *`myfun()`* is an M-function defined in MATLAB.

The function is called only when the command is executed at the MATLAB prompt. The argument of the function `<arg>` called may be MATLAB variables. To include an expression value from the model object, first extract it at the MATLAB prompt, as described in Extracting Results.

The function *`myfun()`* accepts the model object `model` as an input argument as any MATLAB variable.

Extracting Results

Use LiveLink *for* MATLAB to extract at the MATLAB prompt the data computed in the COMSOL model. A suite of wrapper functions is available to perform evaluation operations at the MATLAB prompt.

In this section:

- Extracting Data From Tables
- Extracting Data at Node Points
- Extracting Data at Arbitrary Points
- Evaluating an Expression at Geometry Vertices
- Evaluating an Integral
- Evaluating a Global Expression
- Evaluating a Global Matrix
- Evaluating a Maximum of Expression
- Evaluating an Expression Average
- Evaluating a Minimum of Expression

Extracting Data From Tables

In the table node you can store the data evaluated with the COMSOL built-in evaluation method (see The Numerical Node Syntax).

Use the tag *tbltag* to extract the data stored in the table. Enter:

```
tabl = model.result.table(<tbltag>).getTableData(fullprecision);
```

This creates a `java.lang.string` array `tabl` that contains the data of the table *tbltag*. The size of the array table is *N*x1 where *N* is the number of the table line. *fullprecision* is a Boolean expression to get the data with full precision.

To get the value of a specific row of the table, enter

```
tablline = tabl(<i>);
```

where <*i*> is the number of the desired row. The variable `tablline` is a *M*x1 `java.lang.string` array where *M* is the row number in the table.

To obtain the table header, enter

```
header = model.result.table(<tbltag>).getColumnHeaders;
```

where `header` is a *M*x1 `java.lang.string array`, and *M* is the row number of the table.

To get the table as a double array, use the methods `getReal` and `getImag`:

```
tablReal = model.result.table(<tbltag>).getReal;
tablImag = model.result.table(<tbltag>).getImag;
```

`tablReal` and `tablImag` are available at the MATLAB workspace as NxM arrays where *N* is the number of lines and *M* the number of rows of the table.

To get the table data at a specific row, use the commands:

```
tablRealRow = model.result.table(<tbltag>).getRealRow(<i>);
tablImagRow = model.result.table(<tbltag>).getImagRow(<i>);
```

`tablRealERow` and `tablImagRow` are available at the MATLAB workspace as *N*x1 array where *N* is the number of rows of the table.

See Also

Table in the *COMSOL Java API Reference Guide*

Important

The links to features described outside of this user guide do not work in the PDF, only from within the online help.

Tip

To locate and search all the documentation for this information, in COMSOL, select **Help>Documentation** from the main menu and either enter a search term or look under a specific module in the documentation tree.

Extracting Data at Node Points

The function `mpheval` lets you evaluate expressions on nodes points. The function output is available as a structure in the MATLAB workspace.

Call the function `mpheval` as in this command

```
pd = mpheval(model, <expr>);
```

where <*expr*> is a string cell array that lists the COMSOL expression to evaluate. The expression has to be defined in a COMSOL model object in order to be evaluated.

`pd` is a structure with the following fields:

- `expr` contains the list of names of the expressions evaluated with `mpheval`.
- `d1` contains the value of the expression evaluated. The columns in the data value fields correspond to node point coordinates in columns in the field `p`. In case of several expressions are evaluated in `mpheval`, additional field `d2`, `d3`,... are available.
- `p` contains the node point coordinates information. The number of rows in p is the number of space dimensions.
- `t` contains the indices to columns in `pd.p` of a simplex mesh; each column in `pd.t` represents a simplex.
- `ve` contains the indices to mesh elements for each node points.
- `unit` contains the list of the unit for each evaluated expressions.

SPECIFY THE EVALUATION DATA

The function `mpheval` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution data set to use in the evaluation.

  ```
  pd = mpheval(model, <expr>, 'dataset', <dsettag>);
  ```

  <*dsettag*> is the tag of a solution data set. The default value is the current solution data set of the model. Selection data set such as Cut point, Cut line, Edge, Surface, and so forth, are not supported.

- `selection`, specify the domain selection for evaluation.

  ```
  pd = mpheval(model, <expr>, 'selection', <seltag>);
  ```

  where <*seltag*> is the tag of a selection node to use for the data evaluation. <*seltag*> can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection, the output value is NaN.

- `edim`, specify the element dimension for evaluation.

  ```
  pd = mpheval(model, <expr>, 'edim', edim);
  ```

  where *edim* is either a string or a positive integer such as: `'point'` (`0`), `'edge'` (`1`), `'boundary'` (`2`), or `'domain'` (`3`). The default settings correspond to the model

geometry space dimension. When using a lower space dimension value, make sure that the evaluation point coordinates dimension has the same size.

Note

Use the function `mphevalpoint` to evaluate expressions at geometric points (see Evaluating an Expression at Geometry Vertices).

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters.

  ```
  pd = mpheval(model, <expr>, 'solnum', <solnum>);
  ```

  where `<solnum>` is an integer array corresponding to the inner solution index. `<solnum>` can also be `'end'` to evaluate the expression for the last inner solution. By default the evaluation is done using the last inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps.

  ```
  pd = mpheval(model, <expr>, 'outersolnum', <outersolnum>);
  ```

  where `<outersolnum>` is a positive integer corresponding to the outer solution index. The default setting uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  pd = mpheval(model, <expr>, 't', <time>);
  ```

  where `<time>` is a double array. The default value corresponds to all the stored time steps.

- `phase`, specify the phase in degrees.

  ```
  pd = mpheval(model, <expr>, 'phase', <phase>);
  ```

  where `<phase>` is a double value.

- `pattern`, use Gauss point evaluation.

  ```
  pd = mpheval(model, <expr>, 'pattern','gauss');
  ```

  The default evaluation is performed on the Lagrange points.

OUTPUT FORMAT

The function `mpheval` returns a structure in the MATLAB workspace. You can specify other output data formats.

To only obtain the data evaluation as a double array, set the property `dataonly` to `on`:

```
pd = mpheval(model, <expr>, 'dataonly', 'on');
```

Include the imaginary part in the data evaluation with the property `complexout`:

```
pd = mpheval(model, <expr>, 'complexout', 'on');
```

SPECIFY THE EVALUATION QUALITY

Define function settings to specify the evaluation quality.

- `refine`, specify the element refinement for evaluation.

  ```
  pd = mpheval(model, <expr>, 'refine', <refine>);
  ```

  where *<refine>* is a positive integer. The default value is `1` which set the simplex mesh identical to the geometric mesh.

- `smooth`, specify the smoothing method to enforce continuity on discontinuous data evaluation.

  ```
  pd = mpheval(model, <expr>, 'smooth', smooth);
  ```

  where *smooth* is either `'none'`, `'everywhere'`, or `'internal'` (default). Set the property to `none` to evaluate the data on elements independently, set to `everywhere` to apply the smoothing to the entire geometry, and set to `internal` to smooth the quantity inside the geometry (but no smoothing takes place across borders between domains with different settings). The output with the same data and same coordinates are automatically merged, which means that the output size may differ depending on the smoothing method.

- `recover`, specify the accurate derivative recovery.

  ```
  pd = mpheval(model, <expr>, 'recover', recover);
  ```

  where *recover* is either `'ppr'`, `'pprint'`, or `'off'` (default). Set the property to `ppr` to perform recovery inside domains or set to `pprint` to perform recovery inside domains. Because the accurate derivative processing takes time, the property is disabled by default.

OTHER EVALUATION PROPERTIES

To not use complex-value functions with real inputs, use the property `complexfun`:

```
pd = mpheval(model, <expr>, 'complexfun','off');
```

The default value uses complex-valued functions with real inputs.

Use the property `matherr` to return an error for undefined operations or expressions:

```
pd = mpheval(model, <expr>, 'matherr','on');
```

DISPLAY THE EXPRESSION IN FIGURES

You can display an expression evaluated with `mpheval` in an external figure with the function `mphplot` (see Displaying The Results). The function `mphplot` only supports a MATLAB structure provided by `mpheval` as input.

Extracting Data at Arbitrary Points

The function `mphinterp` evaluates at the MATLAB prompt the result at arbitrary points.

To evaluate an expression at specific point coordinates, call the function `mphinterp` as in the command

```
[d1, ..., dn] = mphinterp(model,{'e1', ..., 'en'},'coord',<coord>);
```

where *e1,...,en* are the COMSOL expressions to evaluate, <*coord*> is a NxM double array, with N the space dimension of the evaluation domain, and M is the number of evaluation points. The output `d1, ..., dn` is a PxM double array, where P is the length of the inner solution.

Alternatively, specify the evaluation coordinates using a selection data set

```
data = mphinterp(model, <expr>, 'dataset', <dsettag>);
```

where <*dsettag*> is a selection data set tag defined in the model, for example, Cut point, Cut Plane, Revolve, and so forth.

SPECIFY THE EVALUATION DATA

The function `mphinterp` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution data set to use in the evaluation.

  ```
  data =
  mphinterp(model,<expr>,'coord',<coord>,'dataset',<dsettag>);
  ```

  <*dsettag*> is the tag of a solution data set. The default value is the current solution data set of the model.

- `selection`, specify the domain selection for evaluation.

```
data =
mphinterp(model,<expr>,'coord',<coord>,'selection',<seltag>);
```

where <*seltag*> is the tag of a selection node to use for the data evaluation. <*seltag*> can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `edim`, specify the element dimension for evaluation.

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,'edim',edim);
  ```

  where *edim* is either a string or a positive integer such as `'point'` (`0`), `'edge'` (`1`), `'boundary'` (`2`), or `'domain'` (`3`). The default settings correspond to the model geometry space dimension. When using a lower space dimension value, make sure that the evaluation point coordinates dimension has the same size.

- `ext`, specify extrapolation control value. This ensures you return data for points that are outside the geometry.

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,'ext',<ext>);
  ```

  where <*ext*> is a double value. The default value is 0.1.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters.

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,solnum',<solnum>);
  ```

  where <*solnum*> is an integer array corresponding to the inner solution index. <*solnum*> can also be `'end'` to evaluate the expression for the last inner solution. By default the evaluation is performed using the last inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps.

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,...
  'outersolnum',<outersolnum>);
  ```

  where <*outersolnum*> is a positive integer corresponding to the outer solution index. The default settings use the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,'t',<time>);
  ```

  where <*time*> is a double array. The default value corresponds to all the stored time steps.

- `phase`, specify the phase in degrees.

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,'phase',<phase>);
  ```

  where *<phase>* is a double value.

OUTPUT FORMAT

The function `mphinterp` returns in the MATLAB workspace a double array. It also supports other output format.

To evaluate several expressions at once, make sure that the same number of output variables are defined as there are expressions specified:

```
[d1, ..., dn] = mphinterp(model,{'e1', ..., 'en'},'coord',<coord>);
```

To extract the unit of the evaluated expression, define an extra output variable

```
[data, unit] = mphinterp(model,<expr>,'coord',<coord>);
```

with `unit` is a 1xN cell array where N is the number of expressions to evaluate.

Include the imaginary part in the data evaluation with the property `complexout`:

```
data = mphinterp(model,<expr>,'coord',<coord>,'complexout','on');
```

To return an error if all evaluation points are outside the geometry, set the property `coorderr` to `on`:

```
data = mphinterp(model,<expr>,'coord',<coord>,'coorderr','on');
```

By default the function returns the value NaN.

SPECIFY THE EVALUATION QUALITY

With the property `recover`, you can specify the accurate derivative recovery

```
data = mphinterp(model,<expr>,'coord',<coord>,'recover', recover);
```

where *recover* is either `'ppr'`, `'pprint'`, or `'off'` (the default). Set the property to `ppr` to perform recovery inside domains or set to `pprint` to apply recovery to all domain boundaries. Because the accurate derivative processing takes time, the property is disabled by default.

OTHER EVALUATION PROPERTIES

Set the `unit` property to specify the unit of the evaluation

```
data = mphinterp(model,<expr>,'coord',<coord>,'unit',<unit>);
```

where *unit* is a cell array with the same size as *expr*.

To not use complex-value functions with real inputs, use the property `complexfun`:

```
data = mphinterp(model,<expr>,'coord',<coord>,'complexfun','off');
```

The default value uses complex-value functions with real inputs.

Use the property `matherr` to return an error for undefined operations or expressions:

```
data = mphinterp(model,<expr>,'coord',<coord>,'matherr','on');
```

Evaluating an Expression at Geometry Vertices

The function `mphevalpoint` returns the result of a given expression evaluated at the geometry vertices

```
[d1,...,dn] = mphevalpoint(model,{'e1',...,'en'});
```

where *`e1,...,en`* are the COMSOL expressions to evaluate. The output `d1, ..., dn` is a NxP double array, where N is the number of evaluation points and P the length of the inner solution.

SPECIFY THE EVALUATION DATA

The function `mphevalpoint` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution data set to use in the evaluation.

  ```
  data = mphevalpoint(model,<expr>,'dataset',<dsettag>);
  ```

  `<dsettag>` is the tag of a solution data set. The default value is the current solution data set of the model.

- `selection`, specify the domain selection for evaluation.

  ```
  data = mphevalpoint(model,<expr>,'selection',<seltag>);
  ```

  where *`<seltag>`* is the tag of a selection node to use for the data evaluation. *`<seltag>`* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection, the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters.

  ```
  data = mphevalpoint(model,<expr>,'solnum',<solnum>);
  ```

  where *`<solnum>`* is an integer array corresponding to the inner solution index. *`<solnum>`* can also be `'end'` to evaluate the expression for the last inner solution. By default the evaluation is done using the last inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps.

  ```
  data = mphevalpoint(model,<expr>,'outersolnum',<outersolnum>);
  ```

  where *<outersolnum>* is a positive integer corresponding to the outer solution index. The default settings use the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphevalpoint(model,<expr>,'t', <time>);
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

Perform a data series operation with the `dataseries` property

```
data = mphevalpoint(model,<expr>,'dataseries', dataseries);
```

where *dataseries* is either `'mean'`, `'int'`, `'max'`, `'min'`, `'rms'`, `'std'`, or `'var'`. Depending on the property value, `mphevalpoint` performs the following operations—mean, integral, maximum, minimum, root mean square, standard deviation, or variance, respectively.

When performing a minimum or maximum operation on the data series, you can specify to perform the operation using the real or the absolute value. Set the property `minmaxobj` to `'real'` or `'abs'`, respectively:

```
data = mphevalpoint(model,<expr>,'dataseries', dataseries,...
'minmaxobj', valuetype);
```

By default *valuetype* is `'real'`.

OUTPUT FORMAT

The function `mphevalpoint` supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable

```
[data,unit] = mphevalpoint(model,<expr>);
```

with `unit` is a 1xN cell array where N is the number of expressions to evaluate.

By default, `mphevalpoint` returns the result s as a squeezed singleton. To get the full singleton set the `squeeze` property to `off`:

```
data = mphevalpoint(model,<expr>,'squeeze','off');
```

Set the property `matrix` to `off` to return the data as a cell array instead of a double array:

```
data = mphevalpoint(model,<expr>,'matrix','off');
```

Evaluating an Integral

Evaluate an integral of expression with the function `mphint2`.

Note

The function `mphint` is now obsolete and will be removed in a future version of the software. If you are using this function in your code, you can now replace it by `mphint2`.

To evaluate the integral of the expression over the domain with the highest space domain dimension call the function `mphint2` as in this command

```
[d1,...,dn] = mphint2(model,'e1',...,'en'},edim);
```

where *`e1,...,en`* are the COMSOL expressions to integrate. The values `d1,...,dn` are returned as a 1xP double array, with P the length of inner parameters. *`edim`* is the integration dimension, which can be `'line'`, `'surface'`, `'volume'`, or an integer value that specifies the space dimension (1, 2, or 3).

SPECIFY THE INTEGRATION DATA

The function `mphint2` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution data set to use in the integration.

  ```
  data = mphint2(model,<expr>,edim,'dataset',<dsettag>);
  ```

  `<dsettag>` is the tag of a solution data set. The default value is the current solution data set of the model.

- `selection`, specify the integration domain.

  ```
  data = mphint2(model,<expr>,edim,'selection',<seltag>);
  ```

  where *`<seltag>`* is the tag of a selection node to use for the data evaluation. *`<seltag>`* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters.

```
data = mphint2(model,<expr>,edim,'solnum',<solnum>);
```

where <*solnum*> is an integer array corresponding to the inner solution index. You can also set the property `solnum` to `'end'` to evaluate the expression for the last inner solution. By default the evaluation is done using the last inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps.

  ```
  data = mphint2(model,<expr>,edim,'outersolnum',<outersolnum>);
  ```

  where <*outersolnum*> is a positive integer corresponding to the outer solution index. The default settings use the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphint2(model,<expr>,edim,'t',<time>);
  ```

  where <*time*> is a double array. The default value corresponds to all the stored time steps.

OUTPUT FORMAT

The function `mphint2` also supports other output format.

To extract the unit of the evaluated expression, define an extra output variable

```
[data,unit] = mphint2(model,<expr>,edim);
```

with `unit` is a 1xN cell array where N is the number of expressions to evaluate.

By default `mphint2` returns the results as a squeezed singleton. To get the full singleton, set the `squeeze` property to `off`:

```
data = mphint2(model,<expr>,edim,'squeeze','off');
```

Set the property `matrix` to `off` to return the data as a cell array instead of a double array:

```
data = mphint2(model,<expr>,edim,'matrix','off');
```

SPECIFY THE INTEGRATION SETTINGS

To specify integration settings such as the integration method, integration order, or axisymmetry assumption using these properties:

- `method`, specify the integration method, which can be either integration or summation.

```
data = mphint2(model,<expr>,edim,'method',method);
```

where *method* can be `'integration'` or `'summation'`. The default uses the appropriate method for the given expression.

- `intorder`, specify the integration order.

  ```
  data = mphint2(model,<expr>,edim,'intorder',<order>);
  ```

  where *order* is a positive integer. The default value is 4.

- `intsurface` or `intvolume`, compute surface or volume integral for axisymmetry models.

  ```
  data = mphint2(model,<expr>,edim,'intsurface','on');
  data = mphint2(model,<expr>,edim,'intvolume','on');
  ```

Evaluating a Global Expression

Evaluate a global expression with the function `mphglobal`.

To evaluate a global expression at the MATLAB prompt, call the function `mphglobal` as in this command

```
[d1,...,dn] = mphglobal(model,{'e1',...,'en'});
```

where *e1,...,en* are the COMSOL global expressions to evaluate. The output values `d1,...,dn` are returned as a Px1 double array, with P the length of inner parameters.

SPECIFY THE EVALUATION DATA

The function `mphglobal` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution data set to use in the evaluation.

  ```
  data = mphglobal(model,<expr>,'dataset',<dsettag>);
  ```

  <*dsettag*> is the tag of a solution data set. The default value is the current solution data set of the model.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters.

  ```
  data = mphglobal(model,<expr>,'solnum',<solnum>);
  ```

  where <*solnum*> is an integer array corresponding to the inner solution index. You can also set the property `solnum` to `'end'` to evaluate the expression for the last inner solution. By default the evaluation is done using the last inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweep.

  ```
  data = mphglobal(model,<expr>,'outersolnum',<outersolnum>);
  ```

  where `<outersolnum>` is a positive integer corresponding to the outer solution index. The default settings uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphglobal(model,<expr>,'t',<time>);
  ```

  where `<time>` is a double array. The default value corresponds to all the stored time steps.

- `phase`, specify the phase in degrees.

  ```
  data = mphglobal(model,<expr>,'phase',<phase>);
  ```

  where `<phase>` is a double value.

OUTPUT FORMAT

The function `mphglobal` also supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable

```
[data,unit] = mphglobal(model,<expr>);
```

with `unit` is a 1xN cell array where N is the number of expression to evaluate.

Include the imaginary part in the data evaluation with the property `complexout`:

```
data = mphglobal(model,<expr>,'complexout','on');
```

OTHER EVALUATION PROPERTIES

Set the `unit` property to specify the unit of the evaluation

```
data = mphglobal(model,<expr>,'unit',<unit>);
```

where `<unit>` is a cell array with the same length as `<expr>`.

To not use complex-value functions with real inputs, use the property `complexfun`:

```
data = mphglobal(model,<expr>,'complexfun','off');
```

The default value uses complex-value functions with real inputs.

Use the property `matherr` to return an error for undefined operations or expressions:

```
data = mphglobal(model,<expr>,'matherr','on');
```

Evaluating a Global Matrix

`mphevalglobalmatrix` evaluates the matrix variable such as S-parameters in a model with several ports activated as a parametric sweep and a frequency-domain study.

To evaluate the global matrix associated to the expression `<expr>`, enter the command:

```
M = mphevalglobalmatrix(model,<expr>);
```

The output data M is a NxN double array, where N is the number of port boundary condition set in the model.

SPECIFY THE EVALUATION DATA

Set the solution data set for evaluation with the property `dataset`

```
data = mphevalglobalmatrix(model,<expr>,'dataset',<dsettag>);
```

where `<dsettag>` is the tag of a solution data.

Evaluating a Maximum of Expression

Use the function `mphmax` to evaluate the maximum of a given expression over an inner solution list.

To evaluate the maximum of the COMSOL expressions *`e1,...,en`* use the command `mphmax` as follows

```
[d1,...,dn] = mphmax(model,{'e1',...,'en'},edim);
```

where *`edim`* is a string to define the element entity dimension: `'volume'`, `'surface'`, or `'line'`. *`edim`* can also be set as a positive integer value (`3`, `2`, or `1` respectively). The output variables d1,...,dn are an NxP array where N is the number of inner solutions and P the number of outer solutions.

SPECIFY THE EVALUATION DATA

The function `mphmax` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution data set to use in the evaluation.

  ```
  data = mphmax(model,<expr>,edim,'dataset',<dsettag>);
  ```

  `<dsettag>` is the tag of a solution data set. The default value is the current solution data set of the model.

- `selection`, specify the domain selection for evaluation.

```
data = mphmax(model,<expr>,edim,'selection',<seltag>);
```

where <*seltag*> is the tag of a selection node to use for the data evaluation. <*seltag*> can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis type: time domain, frequency domain, eigenvalue or stationary with continuation parameters.

  ```
  data = mphmax(model,<expr>,edim,'solnum',<solnum>);
  ```

  where <*solnum*> is an integer array corresponding to the inner solution index. You can also set the property `solnum` to 'end' to evaluate the expression for the last inner solution. By default the evaluation is done using the last inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps.

  ```
  data = mphmax(model,<expr>,edim,'outersolnum',<outersolnum>);
  ```

  where <*outersolnum*> is a positive integer array corresponding to the outer solution index. The default setting uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphmax(model,<expr>,edim,'t',<time>);
  ```

  where <*time*> is a double array. The default value corresponds to all the stored time steps.

OUTPUT FORMAT

The function `mphmax` also supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable

```
[data,unit] = mphmax(model,<expr>,edim);
```

with unit is a 1xN cell array where N is the number of expressions to evaluate.

By default `mphmax` returns the results as a squeezed singleton. To get the full singleton set the `squeeze` property to `off`:

```
data = mphmax(model,<expr>,edim,'squeeze','off');
```

Set the property `matrix` to `off` to return the data as a cell array instead of a double array:

```
data = mphmax(model,<expr>,edim,'matrix','off');
```

Evaluating an Expression Average

Use the function `mphmean` to evaluate the average of a given expression over inner solution lists.

To evaluate the mean of the COMSOL expressions *`e1,...,en`* use the command `mphmean` as follows

```
[d1,...,dn] = mphmean(model,{'e1',...,'en'},edim);
```

where *`edim`* is a string to define the element entity dimension: `'volume'`, `'surface'`, or `'line'`. *`edim`* can also be set as a positive integer value (`3`, `2`, or `1` respectively). The output variables d1,...,dn are an NxP array where N is the number of inner solutions and P the number of outer solutions.

SPECIFY THE EVALUATION DATA

The function `mphmean` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution data set to use in the evaluation.

  ```
  data = mphmean(model,<expr>,edim,'dataset',<dsettag>);
  ```

  `<dsettag>` is the tag of a solution data set. The default value is the current solution data set of the model.

- `selection`, specify the domain selection for evaluation.

  ```
  data = mphmean(model,<expr>,edim,'selection',<seltag>);
  ```

  where *`<seltag>`* is the tag of a selection node to use for the data evaluation. *`<seltag>`* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters.

```
data = mphmean(model,<expr>,edim,'solnum',<solnum>);
```

where <solnum> is an integer array corresponding to the inner solution index. You can also set the property solnum to 'end' to evaluate the expression for the last inner solution. By default the evaluation is done using the last inner solution.

- outersolnum, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps.

  ```
  data = mphmean(model,<expr>,edim,'outersolnum',<outersolnum>);
  ```

  where <outersolnum> is a positive integer array corresponding to the outer solution index. The default setting uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property t:

  ```
  data = mphmean(model,<expr>,edim,'t',<time>);
  ```

  where <time> is a double array. The default value corresponds to all the stored time steps.

OUTPUT FORMAT

The function mphmean also supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable

```
[data,unit] = mphmean(model,<expr>,edim);
```

with unit is a 1xN cell array where N is the number of expressions to evaluate.

By default mphmean returns the results as a squeezed singleton. To get the full singleton set the squeeze property to off:

```
data = mphmean(model,<expr>,edim,'squeeze','off');
```

Set the property matrix to off to return the data as a cell array instead of a double array:

```
data = mphmean(model,<expr>,edim,'matrix','off');
```

SPECIFY THE INTEGRATION SETTINGS

You can specify integration settings such as an integration method or integration order to perform the mean operation. The available integration properties are:

- method, specify the integration method, which can be either integration or summation.

```
data = mphmean(model,<expr>,edim,'method',method);
```

where *method* can be `'integration'` or `'summation'`. The default uses the appropriate method for the given expression.

- `intorder`, specify the integration order.

  ```
  data = mphmean(model,<expr>,edim,'intorder',<order>);
  ```

  where `<order>` is a positive integer. The default value is `4`.

Evaluating a Minimum of Expression

Use the function `mphmin` to evaluate the minimum of a given expression over an inner solution list.

To evaluate the minimum of the COMSOL expressions *e1,...,en* use the command `mphmin` as follows

```
[d1,...,dn] = mphmin(model,{'e1',...,'en'},edim);
```

where *edim* is a string to define the element entity dimension: `'volume'`, `'surface'`, or `'line'`. *edim* can also be set as a positive integer value (`3`, `2`, or `1` respectively). The output variables d1,...,dn are an NxP array where N is the number of inner solutions and P the number of outer solutions.

SPECIFY THE EVALUATION DATA

The function `mphmin` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution data set to use in the evaluation.

  ```
  data = mphmin(model,<expr>,edim,'dataset',<dsettag>);
  ```

  `<dsettag>` is the tag of a solution data set. The default value is the current solution data set of the model.

- `selection`, specify the domain selection for evaluation.

  ```
  data = mphmin(model,<expr>,edim,'selection',<seltag>);
  ```

  where `<seltag>` is the tag of a selection node to use for the data evaluation. `<seltag>` can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters.

  ```
  data = mphmin(model,<expr>,edim,'solnum',<solnum>);
  ```

  where <*solnum*> is an integer array corresponding to the inner solution index. You can also set the property *solnum* to 'end' to evaluate the expression for the last inner solution. By default the evaluation is done using the last inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps.

  ```
  data = mphmin(model,<expr>,edim,'outersolnum',<outersolnum>);
  ```

  where <*outersolnum*> is a positive integer array corresponding to the outer solution index. The default setting uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphmin(model,<expr>,edim,'t',<time>);
  ```

  where <*time*> is a double array. The default value corresponds to all the stored time steps.

OUTPUT FORMAT

The function `mphmin` also supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable

```
[data,unit] = mphmin(model,<expr>,edim);
```

with unit is a 1xN cell array where N is the number of expressions to evaluate.

By default `mphmin` returns the results as a squeezed singleton. To get the full singleton set the `squeeze` property to `off`:

```
data = mphmin(model,<expr>,edim,'squeeze','off');
```

Set the property `matrix` to `off` to return the data as a cell array instead of a double array.

```
data = mphmin(model,<expr>,edim,'matrix','off');
```

Running Models in a Loop

A common use of LiveLink *for* MATLAB is to run models in a loop. As MATLAB offers several functionalities to run loops including conditional statements and error handling, you will see how all these functionality can be used together with the COMSOL Java API syntax to run COMSOL model in loop.

In this section:

- The Parametric Sweep Node
- Running Model in a Loop Using the MATLAB Tools

The Parametric Sweep Node

Using the COMSOL Java API you can run model in loop. See the section Adding a Parametric Sweep in the section *Building Model*s.

Note that using the COMSOL built-in function to run models in loop, you can ensure the model to be saved automatically at each iteration. COMSOL also offers tool to take advantage of clusters architecture.

Running Model in a Loop Using the MATLAB Tools

Use MATLAB tools such as **for** or **while** statements to run your model in a loop. The COMSOL API Java commands can be included in scripts using MATLAB commands. To evaluate such a script you need to have MATLAB connected to a COMSOL server.

To run a model in a loop you do not need to run the entire model M-file commands from scratch. It is recommended to load a COMSOL model in MATLAB and run the loop only over the desired operations. The COMSOL model is automatically updated when running the study node.

You can run a Model M-file from scratch if you need, for instance, to generate the geometry in loop.

Note

The model run inside a MATLAB loop is not automatically saved. Make sure to save the model at each iteration. Use the command `mphsave` to save your model object. If you are not interested in saving the entire model object at each iteration, you can extract data and store it in the MATLAB workspace. See Extracting Results to find the most suitable function to your model.

When running loops in MATLAB, the iteration progress is entirely taking care by MATLAB, only the COMSOL commands are executed in the COMSOL server. You can generate as many nested loops your modeling requires and combine the loop with other MATLAB conditional statement such as `if` and `switch` or error handling statement such as `try/catch`.

Break the loop with `break` or jump to the next loop iteration with `continue`.

Refer to MATLAB help to get more information about the MATLAB commands `for`, `while`, `if`, `switch`, `try/catch`, `break`, and `continue`.

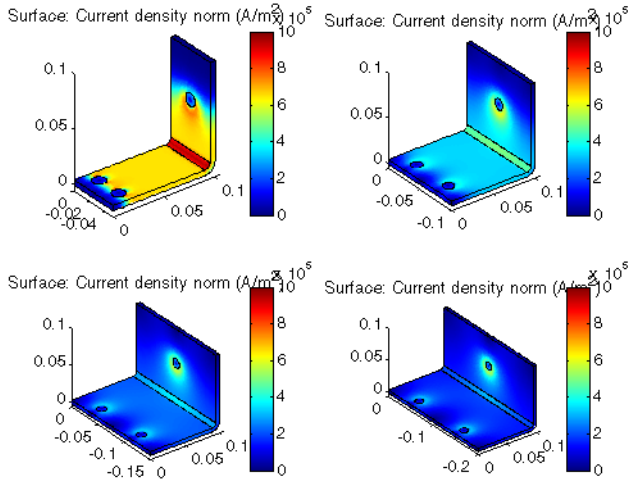
EXAMPLE: GEOMETRY PARAMETRIZATION

In this example you will see how to proceed to geometry parametrization using a MATLAB `for` loop. The model consists of the busbar example available in the COMSOL Multiphysics model, see the *Introduction to COMSOL Multiphysics*.

In this example the loop iterate over the busbar width `wbb`. The solution for each parameter value is displayed using the second plot group defined in the COMSOL model. All the results are plotted in the same figure.

```
model = mphload('busbar');
w = [5e-2 10e-2 15e-2 20e-2];
for i = 1:4
    model.param.set('wbb',w(i));
    model.study('std1').run;
    subplot(2,2,i)
    mphplot(model,'pg2','rangenum',1)
end
```

The resulting figure is shown below:



Running Models in Batch Mode

Use LiveLink *for* MATLAB to models in batch mode. At the MATLAB prompt you can execute commands to set-up the batch job using the COMSOL built-in method or run custom scripts directly from a command line. In this section:

- The Batch Node
- Running A COMSOL M-file In Batch Mode
- Running A COMSOL M-file In Batch Mode Without Display

The Batch Node

Using the COMSOL Java API you can run model in loop. See Adding a Job Sequence.

Running A COMSOL M-file In Batch Mode

To run in batch a M-script that runs COMSOL Model, start COMSOL with MATLAB at a terminal window with the following command:

```
comsol server matlab myscript
```

where myscript is the M-script, saved as myscript.m, that contains the operation to run at the MATLAB prompt.

The COMSOL does not automatically save the model. You need to make sure that the model is saved before the end of the execution of the script. See Loading and Saving a Model.

You can also run the script in batch without the MATLAB desktop and the MATLAB splash. Enter the command below:

```
comsol server matlab myscript -nodesktop -mlnosplash
```

Running COMSOL with MATLAB in batch mode as described in this chapter requires that you have xterm installed on your machine. If this is not the case see Running A COMSOL M-file In Batch Mode Without Display.

Running A COMSOL M-file In Batch Mode Without Display

COMSOL with MATLAB requires that xterm is installed on the machine. If this is not the case as it might be for a computation server, a workaround is to connect manually MATLAB to a COMSOL server with the function `mphstart`.

The steps below describe how to run an M-script that runs a COMSOL model:

1. In a system terminal prompt start a COMSOL server with the command:

   ```
   comsol server &
   ```

2. In the same terminal window change the path to the COMSOL installation directory:

   ```
   cd COMSOL_path/mli
   ```

3. From that location, start MATLAB without display and run the `mphstart` function in order to connect MATLAB to COMSOL:

   ```
   matlab -nodesktop -mlnosplash -r "mphstart; myscript"
   ```

You can get more information about how to connect MATLAB to a COMSOL server in Starting COMSOL with MATLAB on Windows / Mac OSX / Linux.

Working with Matrices

In this section:

- Extracting System Matrices
- Set System Matrices in the Model
- Extracting State-Space Matrices

Extracting System Matrices

Extract the matrices of the COMSOL linearized system with the function `mphmatrix`. To call the function `mphmatrix`, specify a solver node and the list of the system matrices to extract

```
str = mphmatrix(model, <soltag>, 'out', out);
```

where `<soltag>` is the solver node tag used to assemble the system matrices and `out` is a cell array containing the list of the matrices to evaluate. The output data str returned by `mphmatrix` is a MATLAB structure, and the fields correspond to the assembled system matrices.

The system matrices that can be extracted with `mphmatrix` are listed in the table below:

EXPRESSION	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian
D	Damping matrix
E	Mass matrix
NF	Constraint force Jacobian
NP	Optimization constraint Jacobian (*)
MP	Optimization constraint vector (*)
MLB	Lower bound constraint vector (*)
MUB	Upper bound constraint vector (*)
Kc	Eliminated stiffness matrix
Lc	Eliminated load vector

EXPRESSION	DESCRIPTION
Dc	Eliminated damping matrix
Ec	Eliminated mass matrix
Null	Constraint null-space basis
Nullf	Constraint force null-space matrix
ud	Particular solution ud
uscale	Scale vector

(*) Requires the Optimization Module.

SELECTING LINEARIZATION POINTS

The default selection of linearization points for the system matrix assembly is the current solution of the solver node associated to the assembly.

Note

If the linearization point is not specified when calling `mphmatrix`, COMSOL automatically runs the entire solver configuration before assembling and extracting the matrices.

Save time during the evaluation by manually setting the linearization point. Use the `initmethod` property as in this command

```
str = mphmatrix(model, <soltag>, 'out', out, 'initmethod', method);
```

where *`method`* corresponds to the type of linearization point—the initial value expression ('init') or a solution ('sol').

To set the solution to use for the linearization point, use the property `initsol`:

```
str = mphmatrix(model, <soltag>, 'out', out, 'initsol',
<initsoltag>);
```

where *`<initsoltag>`* is the solver tag to use for linearization points. You can also set the `initsol` property to 'zero', which corresponds to using a null solution vector as a linearization point. The default is the current solver node where the assemble node is associated.

For continuation, time-dependent, or eigenvalue analyses you can set the solution number to use as a linearization point. Use the `solnum` property

```
str = mphmatrix(model, <soltag>, 'out', out, 'solnum', <solnum>);
```

where *`<solnum>`* is an integer value corresponding to the solution number. The default value is the last solution number available with the current solver configuration.

See Retrieving Xmesh Information to learn how to get relation between the degrees of freedom information in the matrix system and coordinates or element information.

EXAMPLE: MPHMATRIX

The following illustrates how to use the `mphmatrix` command to extract eliminated system matrices of a stationary analysis and linear matrix system at the MATLAB prompt.

The model consists of a linear heat transfer problem solved on a unit square with a `1e5 W/m^2` surface heat source and temperature constraint. Only one quarter of the geometry is represented in the model. For simplification reasons, the mesh is made of four quad elements.

The commands below set the COMSOL model object:

```
model = ModelUtil.create('Model');

geom1 = model.geom.create('geom1', 2);
geom1.feature.create('sq1', 'Square');
geom1.run;

mat1 = model.material.create('mat1');
def = mat1.materialModel('def');
def.set('thermalconductivity',{'4e2'});

ht = model.physics.create('ht', 'HeatTransfer', 'geom1');
hs1 = ht.feature.create('hs1','HeatSource',2);
hs1.selection.set(1);
hs1.set('Q',1,'1e5');

temp1 = ht.feature.create('temp1','TemperatureBoundary',1);
temp1.selection.set([1 2]);

mesh1 = model.mesh.create('mesh1','geom1');
dis1 = mesh1.feature.create('dis1','Distribution');
dis1.selection.set([1 2]);
dis1.set('numelem','2');
mesh1.feature.create('map1','Map');

std1 = model.study.create('std1');
std1.feature.create('stat','Stationary');
std1.run;
```

To extract the solution vector of the computed solution, run the function `mphgetu` as in this command:

```
U = mphgetu(model);
```

To assemble and extract the eliminated stiffness matrix and the eliminated load vector, set the linearization point to the initial value expression by entering:

```
MA = mphmatrix(model ,'sol1', ...
     'Out', {'Kc','Lc','Null','ud','uscale'},...
     'initmethod','sol','initsol','zero');
```

Solve for the eliminated solution vector using the extracted eliminated system:

```
Uc = MA.Null*(MA.Kc\MA.Lc);
```

Combine the eliminated solution vector and the particular vector:

```
U0 = Uc+MA.ud;
```

Scale back the solution vector:

```
U1 = (1+U0).*MA.uscale;
```

Now compare both solution vector `U` and `U1` computed by COMSOL and by the matrix operation, respectively.

Set System Matrices in the Model

Use the function `mphinputmatrix` to set a linear matrix system to a model.

```
mphinputmatrix(model,<str>,<soltag>,<soltypetag>)
```

The above command set the matrices of a linear system stored in the MATLAB structure <*str*> into the model. The linear system is associated to the solver sequence <*soltag*> and to be solved by the solver <*soltypetag*>.

`mphinputmatrix` only supports the solver types Stationary, Eigenvalue, and Time.

A valid structure <*str*> for a stationary solver includes the following fields:

FIELD	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian

A valid structure *<str>* for a time dependent or an eigenvalue solver includes the following fields:

EXPRESSION	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian
D	Damping matrix
E	Mass matrix

You can also include the Constraint force Jacobian vector, defined in the field NF.

Once the linear system is loaded in the model, you can directly run the solver.

Note

The system matrices are not stored in the model once it is saved in the MPH-format or loaded to the COMSOL Desktop.

EXAMPLE: SET A MODEL WITH A MODIFIED MATRIX SYSTEM

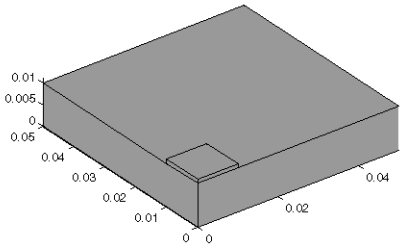
This example deals with heat transfer in solids physics. The geometry and physics settings is already set in the model and saved in the MPH-format. The Model MPH-file comes with the COMSOL installation.

At the MATLAB prompt you will load the model and add an extra line heat source in the model directly in the system matrix, by manually changing the load vector. Then compute the solution of the modified system in COMSOL.

Load the base model MPH-file and display the geometry:

```
model = mphload('model_tutorial_llmatlab.mph');
mphgeom(model)
```

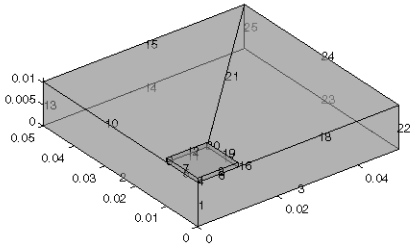
This results in the following MATLAB figure:



Draw a line to be used as line heat source in the model and plot the modified geometry:

```
b1 = model.geom('geom1').feature.create('b1', 'BezierPolygon');
b1.set('p', {'1e-2' '5e-2'; '1e-2' '5e-2'; '1e-2' '1e-2'});
mphgeom(model,'geom1','edgelabels','on','alpha',0.5);
```

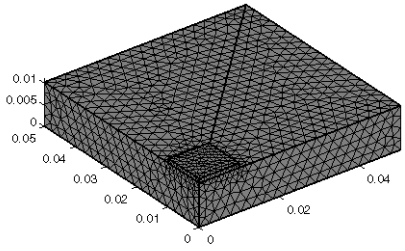
In the figure below you can see that the added line as the index 21:



Generate a mesh with a finer mesh settings:

```
mesh1 = model.mesh('mesh1');
mesh1.feature.create('ftet1', 'FreeTet');
mesh1.feature('size').set('hauto', '3');
```

```
mesh1.run;
mphmesh(model)
```



Set the solver sequence associated to a stationary study node:

```
std1 = model.study.create('std1');
std1.feature.create('stat', 'Stationary');
sol1 = model.sol.create('sol1');
sol1.study('std1');
st1 = sol1.feature.create('st1', 'StudyStep');
st1.set('studystep', 'stat');
v1 = sol1.feature.create('v1', 'Variables');
v1.set('control', 'stat');
sol1.feature.create('s1', 'Stationary');
```

Set the dependent variable discretization with linear shape function:

```
ht = model.physics('ht');
ht.prop('ShapeProperty').set('order_temperature', 1, '1');
```

You can now extract the matrices of the linear system associated to the solver sequence `sol1`:

```
ME = mphmatrix(model,'sol1','Out',{'K' 'L' 'M' 'N'},...
    'initmethod','sol','initsol','zero');
```

To retrieve the degrees of freedom that belong to edge 21, you need first to get the geometric mesh data:

```
[stats,data] = mphmeshstats(model)
```

With the mesh data structure `data`, you can get the element indices that belong to edge 21, use the MATLAB `find` function to list all the indices:

```
elem_idx = find(data.elementity{1}==21)'
```

With the function `mphxmeshinfo` retrieve the finite element mesh information associated to solver sequence sol1:

```
info = mphxmeshinfo(model,'soltag','sol1','studysteptag','v1');
```

In the `info` structure you can get the DOFs indices that belong to the edge element defined with the indices `elem_idx`:

```
edgdofs_idx = [];
for i = 1:length(elem_idx)
    edgdofs_idx = [edgdofs_idx;
info.elements.edg.dofs(:,elem_idx(i))];
end
```

`edgdofs_idx` may contains duplicate DOFs indices. This is because you got the information at the element level; the duplicate indices correspond to the connecting node between two adjacent elements.

First remove the duplicate entities:

```
unique_idx  = unique(edgdofs_idx);
```

You can now edit the load vector for the dofs that belong to edge 21, the total applied power is 50 W.

```
ME.L(unique_idx+1) = 50/length(unique_idx);
```

Now that the linear system has been modified you can set it back in the model:

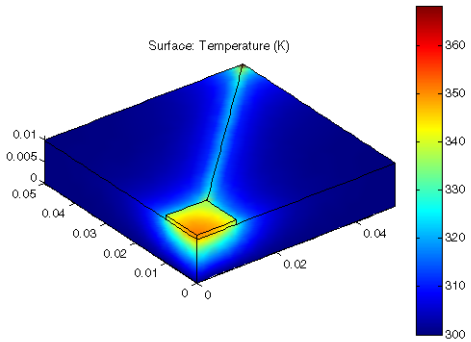
```
mphinputmatrix(model,ME,'sol1','s1')
```

Compute the solution of the added system

```
model.sol('sol1').runAll;
```

Display the solution.

```
pg1 = model.result.create('pg1', 'PlotGroup3D');
pg1.feature.create('surf1', 'Surface');
mphplot(model,'pg1','rangenum',1)
```



Extracting State-Space Matrices

Use state-space export to create a linearized state-space model corresponding to a COMSOL Multiphysics model. You can export the matrices of the state-space form directly to the MATLAB workspace with the command `mphstate`.

THE STATE-SPACE SYSTEM

A state-space system is the mathematical representation of a physical model. The system consistent in an ODE linking input, output, and state-space variable. A dynamic system can be represented with the following system:

$$\begin{cases} \dfrac{dx}{dt} = Ax + Bu \\ y = Cx + Du \end{cases}$$

An alternative representation of the dynamic system is:

$$\begin{aligned} Mc\dot{x} &= McAx + McBu \\ y &= Cx + Du \end{aligned}$$

This form is more suitable for large systems because the matrices M_C and M_A usually become much more sparse than A.

If the mass matrix M_C is small, it is possible to approximate the dynamic state-space model with a static model, where $M_C = 0$:

$$y = (D - C(McA)^{-1}McB)u$$

Let *Null* be the PDE constraint null-space matrix and ud a particular solution fulfilling the constraints. The solution vector U for the PDE problem can then be written

$$U = \text{Null}x + ud + u0$$

where $u0$ is the linearization point, which is the solution stored in the sequence once the state-space export feature is run.

EXTRACT STATE-SPACE MATRICES

The function `mphstate` requires that the input variables, output variables, and the list of the matrices to extract in the MATLAB workspace are all defined

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out);
```

where `<soltag>` is the solver node tag used to assemble the system matrices listed in the cell array `out`, and `<input>` and `<output>` are cell arrays containing the list of the input and output variables, respectively.

The output data str returned by `mphstate` is a MATLAB structure and the fields correspond to the assembled system matrices.

The input variables need to be defined as parameters in the COMSOL model. The output variables are defined as domain point probes or global probes in the COMSOL model.

The system matrices that can be extracted with `mphstate` are listed in the table below:

EXPRESSION	DESCRIPTION
`MA`	McA matrix
`MB`	McB matrix
`A`	A matrix
`B`	B matrix
`C`	C matrix
`D`	D matrix
`Mc`	Mc matrix
`Null`	Null matrix
`ud`	ud vector
`x0`	x0 vector

To extract sparse matrices set the property sparse to on:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'sparse', 'on');
```

To keep the state-space feature node, set the property `keepfeature` to on:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'keepfeature', 'on');
```

SET LINEARIZATION POINTS

`mphstate` uses linearization points to assemble the state-space matrices. The default linearization point is the current solution provided by the solver node, to which the state-space feature node is associated. If there is no solver associated to the solver configuration, a null solution vector is used as a linearization point.

Note

The linearization point needs to be a steady-state solution.

You can manually select which linearization point to use. Use the initmethod property to select a linearization point

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'initmethod', method);
```

where *method* corresponds to the type of linearization point—the initial value expression ('init') or a solution ('sol').

To set the solution to use for the linearization point, use the property initsol:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'initsol', <initsoltag>);
```

where <*initsoltag*> is the solver tag to use for a linearization point. You can also set the `initsol` property to 'zero', which corresponds to using a null solution vector as a linearization point. The default is the current solver node where the assemble node is associated.

For continuation, time-dependent, or eigenvalue analyses you can set which solution number to use as a linearization point. Use the solnum property

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'solnum', <solnum>);
```

where <*solnum*> is an integer value corresponding to the solution number. The default value is the last solution number available with the current solver configuration.

EXAMPLE

To illustrate how to use the `mphstate` function to extract the state-space matrices of the model **heat_transient_axi** from the *COMSOL Multiphysics Model Library*. To be able to extract the state-space matrices you need to do some modifications to an existing model. First, create a parameter `T0` that is set as the external temperature:

```
model = mphload('heat_transient_axi');
model.param.set('Tinput','1000[degC]');
model.physics('ht').feature('temp1').set('T0', 1, 'Tinput');
```

Then create a domain point probe:

```
pdom1 = model.probe.create('pdom1', 'DomainPoint');
pdom1.model('mod1');
pdom1.setIndex('coords2','0.28',0,0);
pdom1.setIndex('coords2','0.38',0,1);
```

Extract the matrices of the state-space system using `Tinput` as an input variable and the probe `mod1.ppb1` as an output variable:

```
M = mphstate(model,'sol1','out',{'Mc' 'MA' 'MB' 'C' 'D'},...
         'input','T0', 'output', 'mod1.ppb1');
```

Compute the state-space system with the extracted matrices:

```
T0 = 273.15;
Tinput = 1273.15-T0;

opt = odeset('mass', M.Mc);
func = @(t,x) M.MA*x + M.MB*Tinput;
[t,x] = ode23s(func, [0:10:190], zeros(size(M.MA,1),1), opt);
y = M.C*x';
y = y+T0;
```

Compare the solution computed with the state-space system and the one computed with COMSOL:

```
plot(t,y)
hold on
Tnum = mphinterp(model,'T','coord',[0.28;0.38],'t',t);
plot(t,Tnum,'r+')
```

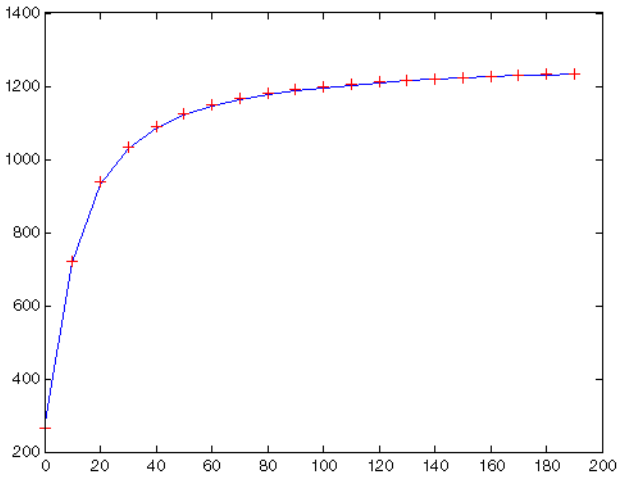


Figure 4-1: Temperature distribution computed with the state-space system (blue line) and COMSOL Multiphysics (red marker).

Extracting Solution Information and Solution Vector

In this section:

- Obtaining Solution Information
- Extracting Solution Vector

Obtaining Solution Information

Get the solution object information with the function `mphsolinfo`. Specify only the model object to obtain the information of the default solution object:

```
info = mphsolinfo(model)
```

Note

The function `mphsolinfo` replaces the function `mphgetp`.

SPECIFYING THE SOLUTION OBJECT

To retrieve the information of a specific solution object, set the solname property with the solver tag *`soltag`* associated to the solution object:

```
info = mphsolinfo(model, 'soltag', <soltag>);
```

If there are several solution data sets attached to the solver, for example, solution data sets with different selections, specify the data set to use to get the solution object information with the `dataset` property

```
info = mphsolinfo(model, 'dataset', <dsettag>);
```

where *`dsettag`* the tag of the solution data set to use.

OUTPUT FORMAT

The output info is a MATLAB structure. The default fields available in the structure are listed in the following table:

FIELDS	DESCRIPTION
`soltag`	Tag of the solver associated to the solution object
`study`	Tag of the study associated to the solution object
`size`	Size of the solution vector
`nummesh`	Number of mesh in the solution (for automatic remeshing)
`sizes`	Size of solution vector and inner parameters for each mesh
`soltype`	Solver type
`solpar`	Parameter name
`sizesolvals`	Length of parameter list
`solvals`	Inner parameter value
`paramsweepnames`	Outer parameter name
`paramsweepvals`	Outer parameter value
`batch`	Batch information
`dataset`	Tag of the solution data set associated to the solution object

To get the information about the number of solutions, set the property `nu` to `on`:

```
info = mphsolinfo(model, 'nu', 'on');
```

The info structure is added with the following fields:

FIELDS	DESCRIPTION
`NUsol`	Number of solutions vectors stored
`NUreacf`	Number of reaction forces vectors stored
`NUadj`	Number of adjacency vectors stored
`NUfsens`	Number of functional sensitivity vectors stored
`NUsens`	Number of forward sensitivity vectors stored

The `batch` field is a a structure including the following fields:

BATCH FIELDS	DESCRIPTION
`type`	The type of batch
`psol`	Tag of the associated solver node

BATCH FIELDS	DESCRIPTION
`sol`	Tag of the stored solution associated to psol
`seq`	Tag of the solver sequence associated to psol

Extracting Solution Vector

Extract the solution vector with the function `mphgetu`, enter

```
U = mphgetu(model);
```

where U is an Nx1 double array, with N the number of degrees of freedom of the COMSOL model.

See Also

You can refer to the function `mphxmeshinfo` to receive the dof name or the node coordinates in the solution vector, see Retrieving Xmesh Information.

SPECIFYING THE SOLUTION

Change the solver node to extract the solution vector with the property `solname`

```
U = mphgetu(model, 'soltag', <soltag>);
```

where <*soltag*> is the tag of the solver node.

For solver settings that compute for several inner solutions, select the inner solution to use with the `solnum` property

```
U = mphgetu(model, 'solnum', <solnum>);
```

where <*solnum*> a positive integer vector that corresponds to the solution number to use to extract the solution vector. For time-dependent and continuation analyses, the default value for the solnum property is the last solution number. For an eigenvalue analysis, it is the first solution number.

A model can contain different types of solution vectors—the solution of the problem, the reaction forces vector, the adjoint solution vector, the functional sensitivity vector, or the forward sensitivity. In `mphgetu`, you can specify the type of solution vector to extract with the `type` property

```
U = mphgetu(model, 'type', type);
```

where *type* is one of the following strings 'sol', 'reacf', 'adj', or 'sens' to extract the solution vector, the reaction forces, the functional sensitivity, or the forward sensitivity, respectively.

OUTPUT FORMAT

`mphgetu` returns the default the solution vector. Get the time derivative of the solution vector Udot by adding a second output variable:

```
[U, Udot] = mphgetu(model);
```

In case the property `solnum` is set as a 1x M array and the solver node only uses one mesh to create the solution, the default output is an NxM array, where N is the number of degrees of freedom of the model. Otherwise, the output `U` is a cell array that contains each solution vector. If you prefer to have the output in a cell array format, set the property matrix to off:

```
U = mphgetu(model, 'solnum', <solnum>, 'matrix', 'off');
```

Retrieving Xmesh Information

Use LiveLink *for* MATLAB to retrieve at the MATLAB workspace low level information of the COMSOL finite element model.

In this section:

- The Extended Mesh (Xmesh)
- Extracting Xmesh Information

The Extended Mesh (Xmesh)

The extended mesh (xmesh) is the finite element mesh that is used to compute the solution. This contains the information about elements, nodes, and degrees of freedom such as DOF names, position of the nodes in the assembled matrix system, or how elements and nodes are connected.

Extracting Xmesh Information

The function `mphxmeshinfo` returns the extended mesh information. To get the xmesh information of the current solver and mesh node, enter the command

```
info = mphxmeshinfo(model);
```

where info is a MATLAB structure that contains the fields listed in this table:

FIELDS	DESCRIPTION
`soltag`	Tag of the solver node
`ndofs`	Number of degrees of freedom
`fieldnames`	List of field variables names
`fieldndofs`	Number of degrees of freedom for each field variables
`meshtypes`	List of the mesh type
`geoms`	Tag of the geometry node used in the model
`dofs`	Structure containing the dofs information
`nodes`	Structure containing the nodes information
`elements`	Structure containing the elements information

The `dofs` substructure contains the fields listed in the following table:

FIELDS	DESCRIPTION
`geomnums`	Index of the geometry tag for each dofs
`coords`	Coordinates of the dofs
`nodes`	Nodes index of the dofs
`dofnames`	Variable names
`nameinds`	Variable names index of the dofs

The `nodes` substructure contains the fields listed in this table:

FIELDS	DESCRIPTION
`coords`	Nodes coordinates
`dofnames`	Variable names
`dofs`	NxM array containing the index (0-based) of the dofs for each node. N being the length of dofnames and M the number of nodes

The `element` substructure contains the fields listed in this table:

FIELDS	DESCRIPTION
`meshtypes`	List of the type of mesh available
`type`	Substructure containing the information of element of type *`type`*

The *`type`* substructure lists the information for each element. The possible mesh types are `vtx`, `edg`, `quad`, `tri`, `quad`, `tet`, `hex`, `prism`, and `pyr`. The substructure *`type`* contains the fields listed in this table:

FIELDS	DESCRIPTION
`localcoords`	Local nodes coordinates
`localdofcoords`	Local dofs coordinates
`localdofnames`	Names of the local dofs
`nodes`	Nodes index for each element
`dofs`	Dofs index for each element

SPECIFY THE INFORMATION TO RETRIEVE

To specify the solver node to retrieve the xmesh information, set the property `solname` as in this command

```
info = mphxmeshinfo(model, 'soltag', <soltag>);
```

where <*soltag*> is the tag of the solver used to extract the xmesh information.

To retrieve the xmesh information for a specific study step node, specify it with the property studysteptag:

```
info = mphxmeshinfo(model, 'studysteptag', <studysteptag>);
```

where <*studysteptag*> is the tag of either a compiled equation node or a variable node.

In case several mesh cases have been used by a specific solver, for example, with an automatic remeshing procedure, you can specify which mesh case to use to get the discretization information

```
info = mphxmeshinfo(model, 'meshcase', <meshcase>);
```

where <*meshcase*> is the mesh case number or the tag of the mesh case.

Navigating the Model

The model object contains all the finite element model settings. To retrieve the model information you can navigate in the model object using a graphical user interface or directly at the MATLAB prompt. Learn how to get the list of predefined expressions available for a given model and how to extract the value of these expressions and also the properties of the method used in the model.

In this section:

- Navigating the Model Object Using a GUI
- Navigating The Model Object At The Command Line
- Finding Model Expressions
- Getting Feature Model Properties
- Getting Model Expressions
- Getting Selection Information

Navigating the Model Object Using a GUI

The usual approach to navigate through the model object in a graphical user interface is to load the model object at the COMSOL Desktop. Then transfer the model object from the COMSOL server to the COMSOL Desktop as in the section Exchanging Models Between MATLAB and the COMSOL Desktop.

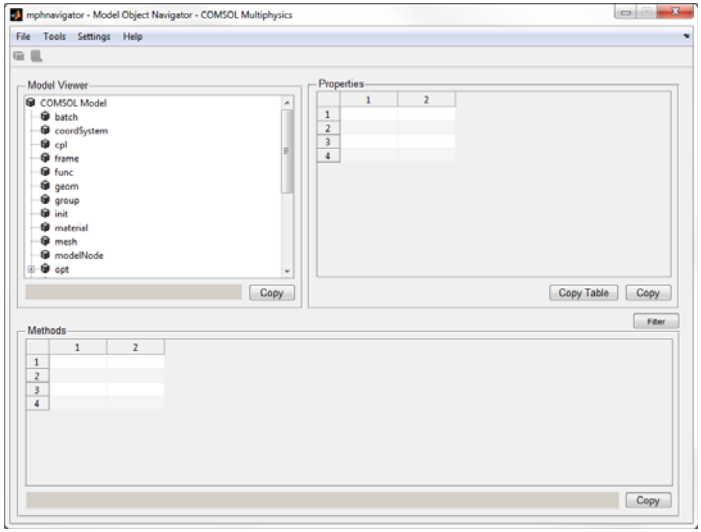
An alternative approach is to call the function `mphnavigator` that displays the model object information in a MATLAB graphical user interface. To run the function type at the MATLAB prompt, enter the command:

```
mphnavigator
```

Note

Prior to calling `mphnavigator`, make sure that the MATLAB object linked the COMSOL model object has the same name. No other name is currently supported.

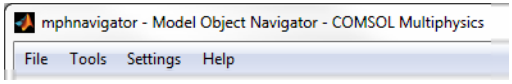
This command pops-up a MATLAB GUI as in the figure below:



Important

If a new model object is created with the MATLAB object name `model`, restart **mphnavigator** in order to have the updated model information.

THE MENU BAR ITEMS



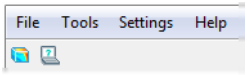
The **mphnavigator** GUI menu bar has the following menu:

- The **File** menu, where the current model object can be saved in the MPH-format, a new model object can be opened, and the **mphnavigator** window can be closed.
- The **Tools** menu lists the navigation tools available for the model object. **Search** is a shortcut to the command **mphsearch** that starts a GUI to search expressions or tags in the model object (see also Finding Model Expressions). **Solutions** starts a GUI to display the solution object available in the COMSOL model object. **Show Errors** lists the error or warning nodes available in the model object (see Handling Errors and Warnings).

- The **Settings** menu only contains the **Advanced** options. Click it to select or deselect the advanced model object methods that are displayed in the **Model Viewer** tree.
- The **Help** menu.

THE SHORTCUT ICON

Just under the menu bar are two shortcut buttons—**Plot** and **Help**.

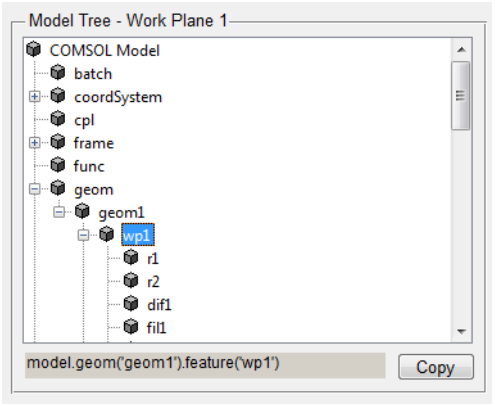


These buttons are grayed if no method has been selected in the **Model Tree** section.

The **Plot** button displays the geometry, the mesh, or a plot group in a MATLAB figure.

The **Help** button displays the page of the *COMSOL Java API Reference Guide* of the corresponding method in your default web browser.

THE MODEL TREE SECTION

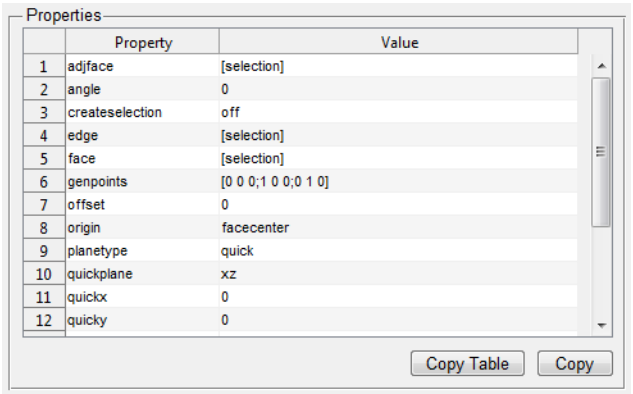


The **Model Tree** section has the list of the nodes of the model object. Use the scroll bar to the right to scroll down the list and click the + icon to expand the model object feature nodes.

When a feature node is selected, its associated command is listed just beneath the model tree. Click **Copy** to copy syntax to the clipboard, and then paste it in your script.

Note	The **Model Tree** list is slightly different to the **Model Builder** list available in the COMSOL Desktop. This is because `mphnavigator` displays all feature nodes and does not use the same filter as in the COMSOL Desktop to order the available feature nodes.

THE PROPERTIES SECTION

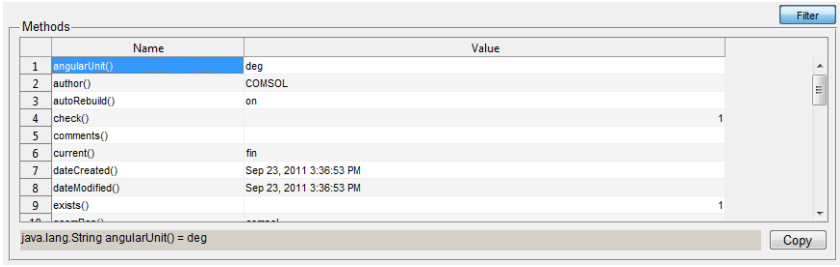


The **Properties** section lists the properties of a selected feature node and the associated values.

Click **Copy Table** to copy the entire properties table to the clipboard, then paste into a text or spreadsheet editor.

Click **Copy** to copy a selected cell in the properties table.

THE METHODS SECTION



The **Methods** section lists all the methods associated to the feature node selected in the Model Tree section.

Click **Filter** to filter the reduce the methods list to the one that returns simple information.

Select a method in the list to get its associated syntax at the button of the **Methods** section. Use the **Copy** button to copy the syntax to the clipboard.

Navigating The Model Object At The Command Line

Retrieve model object information, such as tags for nodes and subnodes of a COMSOL model object at the MATLAB prompt, using the command `mphmodel`.

To get the list of the main feature nodes and the tags of the model object `model`, enter the command:

```
mphmodel(model)
```

To list the subfeature of the node type `model.feature` enter the command:

```
mphmodel(model.feature)
```

To list the subfeature node of the feature node `model.feature(<ftag>)`, enter:

```
mphmodel(model.feature(<ftag>))
```

Use the flag `-struct` to return the model object information to MATLAB structure:

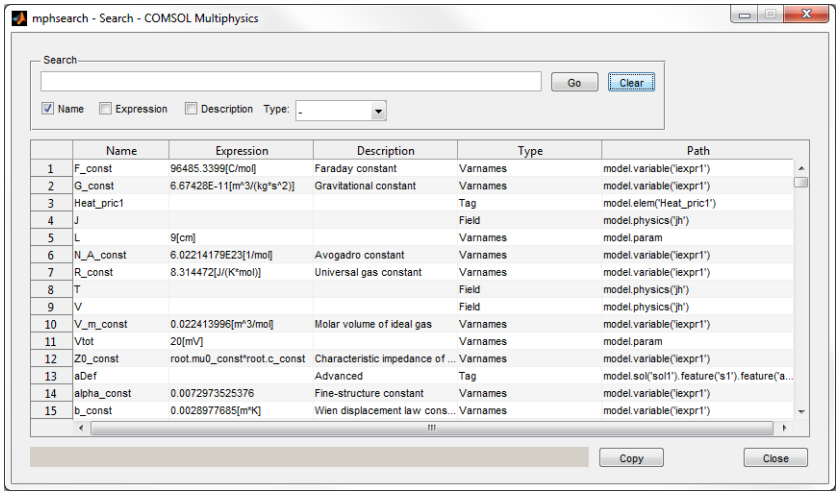
```
str = mphmodel(model.feature,'-struct')
```

str is a MATLAB structure and the fields consist of each feature node associated to the node model.*feature*.

Finding Model Expressions

Each model object contains predefined expressions that depend on the physics interface used in the model.

The function `mphsearch` starts a MATLAB GUI that displays the list of all the expressions, constants, solution variables, or parameters available in the model object.



	Name	Expression	Description	Type	Path
1	F_const	96485.3399[C/mol]	Faraday constant	Varnames	model.variable('iexpr1')
2	G_const	6.67428E-11[m^3/(kg*s^2)]	Gravitational constant	Varnames	model.variable('iexpr1')
3	Heat_pric1			Tag	model.elem('Heat_pric1')
4	J			Field	model.physics('jh')
5	L	9[cm]		Varnames	model.param
6	N_A_const	6.02214179E23[1/mol]	Avogadro constant	Varnames	model.variable('iexpr1')
7	R_const	8.314472[J/(K*mol)]	Universal gas constant	Varnames	model.variable('iexpr1')
8	T			Field	model.physics('jh')
9	V			Field	model.physics('jh')
10	V_m_const	0.022413996[m^3/mol]	Molar volume of ideal gas	Varnames	model.variable('iexpr1')
11	Vtot	20[mV]		Varnames	model.param
12	Z0_const	root.mu0_const*root.c_const	Characteristic impedance of ...	Varnames	model.variable('iexpr1')
13	aDef		Advanced	Tag	model.sol('sol1').feature('s1').feature('a...
14	alpha_const	0.0072973525376	Fine-structure constant	Varnames	model.variable('iexpr1')
15	b_const	0.0028977685[m*K]	Wien displacement law cons...	Varnames	model.variable('iexpr1')

The table has the following for each entry, the:

- **name** of the expression,
- **expression** as it is set in the property value,
- **description** if there is one set for the expression,
- **type** of the expression, and the
- **path** in the model object.

The **Search** section has a searching tool to filter the list. Enter any string in the edit field and select where to search this string—in the name, the expression, or the description of the table entry. You can also select the type you want to list. The expression type can be **Equation**, **Field**, **Tag**, **VarNames**, or **Weak**.

Click **Go** to display the result of the search. Click **Clear** to clear the search settings.

Click **Copy** to copy any entry of the table to the clipboard.

Click **Close** to close the **mphsearch** window.

Getting Feature Model Properties

Use the command `mphgetproperties` to extract at the MATLAB prompt the properties of a specified node of the model object. Use the command

```
prop = mphgetproperties(model.feature)
```

where expr is a MATLAB structure that lists all the properties and the value of the feature node model.*feature*.

Getting Model Expressions

Use the command `mphgetexpressions` to get at the MATLAB prompt the expressions and the descriptions of a specified node of the model object. Use the command

```
expr = mphgetexpressions(model.feature)
```

where model.*feature* is the node to get the expressions from and expr is an Nx3 cell array where N is the number of expressions for this node.

Getting Selection Information

Use the function `mphgetselection` to retrieve the model selection information

```
str = mphgetselection(model.selection(<seltag>))
```

where *seltag* is the tag a selection node defined in the model object. The output str is a MATLAB structure with the following fields:

- **dimension**, the space dimension of the geometry entity selected,
- **geom**, the tag of the geometry node used in the selection,
- **entities**, the list of the entity indexes listed in the selection, and
- **isGlobal**, Boolean value to indicate if the selection is global or not.

Handling Errors and Warnings

In this section:

- Errors and Warnings
- Using MATLAB Tools To Handle COMSOL Exceptions
- Displaying Warnings and Errors in the Model

Errors and Warnings

COMSOL Multiphysics reports these types of problems:

- Errors, which prevents the program from completing a task, and
- Warnings, which are problems that do not prevent the completion of a task but that might affect the accuracy or other aspects of the model.

For both errors and warnings a message is stored in a separate node located just below the problematic model feature node.

In case of errors, a Java Exception is thrown to MATLAB, which also breaks the execution of the script.

Using MATLAB Tools To Handle COMSOL Exceptions

When running a model that returns an error in MATLAB, the execution of the script is automatically stopped. You can use MATLAB tools to handle exceptions and prevent the script from breaking. Use the `try` and `catch` MATLAB statements to offer alternatives to a failed model.

In a loop, for example, use the `try` and `catch` statements to continue to the next iteration. For automatic geometry or mesh generation you can use it to set the model properties with alternative values that circumvent the problem.

Displaying Warnings and Errors in the Model

Use the command `mphshowerrors` to search in a given model object for warning or error nodes.

To display the error and warning messages and their location in the model object type the command:

```
mphshowerrors(model)
```

Alternatively `mphshowerrors` can also return the error and warning information in a MATLAB variable:

```
str = mphshowerrors(model)
```

where str is an *N*x2 cell array, with *N* the number of error and warning nodes that contain the model object. `str{i,1}`, which contains the location in the model of the *i*:th error/warning message and `str{i,2}` contains the message of the ith error/warning message.

Improving Performance for Large Models

Memory management is a key to successful modeling. In COMSOL Multiphysics the finite element model can store a large amount of data depending on the complexity of the model. Exchanging such a large amount of data between MATLAB and the COMSOL server can be problematic in terms of memory or computational time. This section discusses the model settings if you are experiencing memory problems or slowness of command execution.

- Setting Java Heap Size
- Disabling Model Feature Update
- Disabling The Model History

Setting Java Heap Size

COMSOL stores the data in Java. If you are experiencing memory problems during meshing, postprocessing operations, or when exchanging data between the COMSOL server and MATLAB, this may mean that the Java Heap size is set with too low a value.

Note

Increasing the memory allocated for the Java process necessarily decreases the memory available for the solver.

THE COMSOL SERVER JAVA HEAP SIZE

You can access the Java Heap size settings for the COMSOL server process in the `comsolserver.ini` file that can be found in the `COMSOL43a/bin/<arch>` directory. `<arch>` correspond to the architecture of the machine where the COMSOL server is running. Edit the file with a text editor, the Java heap settings are set as:

```
-Xss4m
-Xms40m
-Xmx1024m
-XX:MaxPermSize=256m
```

The values are given in Mb, modify these value to satisfy the model requirements.

THE MATLAB JAVA HEAP SIZE

To modify the Java heap size you need to edit the java.opts file available under the COMSOL with MATLAB start-up directory. The java.opts file is stored by default with the following settings:

```
-Xss4m
-Xmx768m
-XX:MaxPermSize=256m
```

The values are given in Mb, modify these value to satisfy the model requirements.

To modify the MATLAB Java Heap size the `java.opts` file has to be stored at the MATLAB start-up directory. This is the case when starting COMSOL with MATLAB.

In case you are manually connecting MATLAB with a COMSOL server, make sure you have the java.opts at the MATLAB start-up directory.

Disabling Model Feature Update

For models that contain a large amount of physics feature nodes, it may help to deactivate the model feature update while implementing the model object. By default COMSOL updates the expression value for every feature node in the model, which can take some time.

To disable the feature model update enter:

```
model.disableUpdates(true);
```

You need to enable the feature update prior to computing the solution unless the model expressions would not be updated according to the model settings. This is also necessary if you are building a geometry or a mesh that depends on expressions.

To enable the feature model update, enter:

```
model.disableUpdates(false);
```

Disabling The Model History

If you are experiencing a slow down of the operation run in loop as the number iteration increases. A possible reason is that the model history uses a significant amount memory that can no longer be accessible by the COMSOL operation. You can disable the history recording to keep the model history information low.

To disable the model history, enter the command:

```
model.hist.enable
```

When the model history is disabled you no longer see the commands used to set up the model when saving it as an M-file.

If you load the model object with the function `mphload`, it automatically disables the model history.

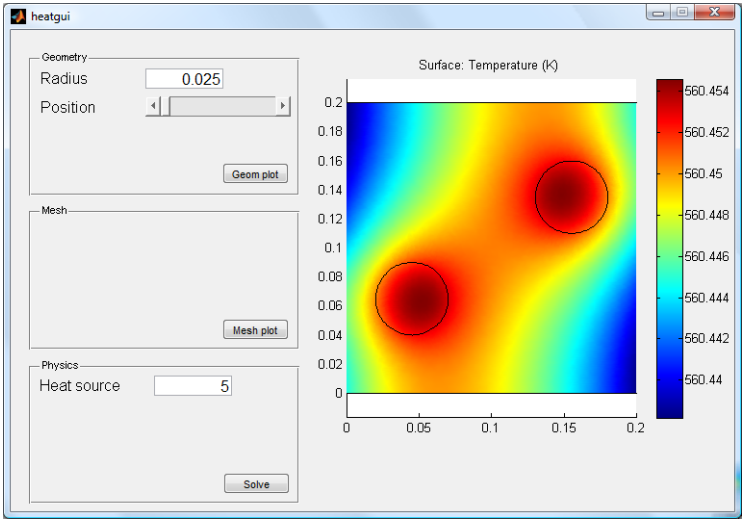
To activate the model history, enter the command:

```
model.hist.disable
```

Creating a Custom GUI

You can use the MATLAB **guide** functionality to create a GUI and connect the interface to a COMSOL model object. Each operation at the GUI sets the value of a MATLAB variable or calls a MATLAB command. As described in this guide, you can call commands at the MATLAB prompt to set up a COMSOL model object or set MATLAB variables in the COMSOL model object.

The figure below illustrates a GUI made in MATLAB and linked to a COSMOL model object.



The simplified GUI only allows the user to compute a heat transfer problem on a given geometry. The user can only change the radius and the position of the bottom circle geometry. The heat source applied to the bottom circle is also defined by the user.

The button executes the building operation of the geometry and mesh. Another button executes the computation of the solution.

COMSOL 3.5a Compatibility

COMSOL makes an effort to be backward compatible: you can load model MPH-files created in COMSOL Multiphysics 3.5a and later versions in COMSOL Multiphysics 4.3a.

When going from version 3.5a to version 4, a major revision was made to the MATLAB interface. This revision was made to reflect changes made to the new user interface and to support parameterized geometry operations. As a result, a new MATLAB interface syntax is used in today's version of COMSOL Multiphysics and its add-on product LiveLink *for* MATLAB.

In order to assist in the conversion process, a special compatibility mode was created to facilitate the new syntax. This compatibility mode, together with the LiveLink *for* MATLAB function **mphv4**, is no longer supported as of COMSOL Multiphysics 4.3.

If you wish to convert a model defined with an M-file created with version 3.5a to the version 4.3a format, we recommend the following procedure:

1. Run the M-file using COMSOL Multiphysics 3.5a and save the model, using **flsave**, as an MPH-file.
2. Load the model into COMSOL Multiphysics 4.3a and verify that the model settings have been translated correctly. In addition, verify that the model can be meshed and solved.
3. Select **File>Reset history**.
4. Select **File>Save as Model M-file**.

The saved M-file can now be tested if you start the current version of COMSOL Multiphysics with MATLAB.

If you have any problems with this conversion process, please contact COMSOL's technical support team at support@comsol.com, or your local COMSOL representative.

5

Calling MATLAB Functions

This section introduces you to the MATLAB function callback from the COMSOL Desktop and COMSOL model object.

In this chapter:

- The MATLAB Function Feature Node

The MATLAB Function Feature Node

MATLAB functions are global in scope and you can use them in a model to define model settings such as parameters, material properties, and boundary conditions.

When running the model, COMSOL automatically starts a MATLAB process that evaluates the function and returns the value to the COMSOL model.

Note

To call a MATLAB function from within the model you do not need to start COMSOL with MATLAB—starting the COMSOL Desktop is sufficient. The MATLAB process starts automatically to evaluate the function.

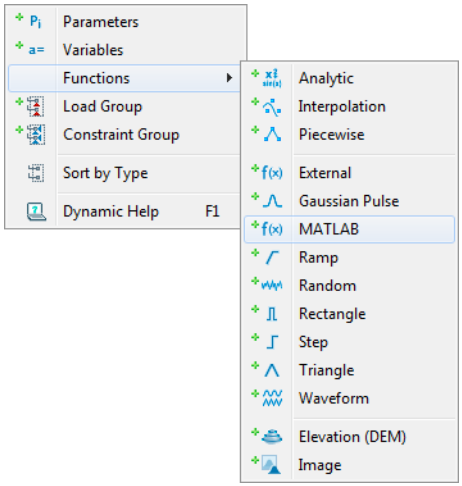
- Defining a MATLAB Function in the COMSOL Model
- Adding a MATLAB Function with the COMSOL API Java Syntax

Defining a MATLAB Function in the COMSOL Model

ADDING THE MATLAB FUNCTION NODE

To evaluate a MATLAB function from within the COMSOL model you need to add a MATLAB node in the model object where the function name, the list of the arguments, and, if required, the function derivatives are defined.

To add a MATLAB function node, right-click the **Global Definitions** node and select **Functions>MATLAB**.

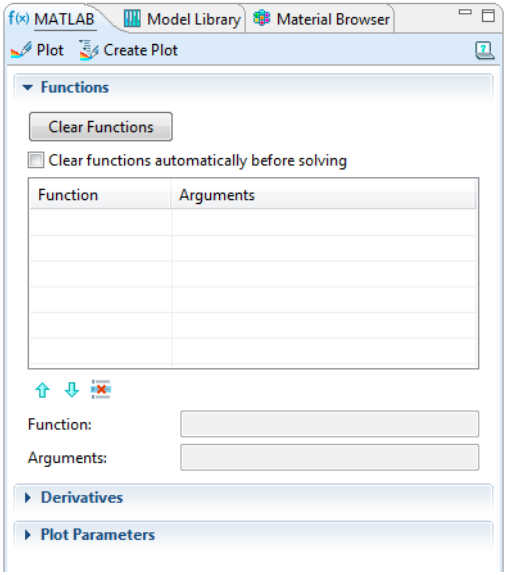


The settings window of the **MATLAB** node has these sections:

- **Functions** where you declare the name of the MATLAB functions and their arguments.
- **Derivatives** where you define the derivative of the MATLAB functions with respect to all function arguments.
- **Plot Parameters** where you can define the limit of the arguments value in order to display the function in the COMSOL Desktop Graphics window.

DEFINING THE MATLAB FUNCTION

This figure illustrates the **MATLAB** settings window:



Under **Functions**, define the function name and the list of the function arguments.

In the table columns and rows, enter the **Function** name and the associated function **Arguments**. The table supports multiple function definitions. Define several functions in the same table or add several **MATLAB** nodes, as you prefer.

PLOTTING THE FUNCTION

Click the **Plot** button ( ) to display a plot of the function.

Click the **Create Plot** button ( ) to create a plot group under the **Results** node.

To plot the function you first need to define limits for the arguments. Expand the **Plot Parameters** section and enter the desired value in the **Lower limit** and **Upper limit** columns. In the **Plot Parameters** table the number of rows correspond to the number of input arguments of the function. The first input argument corresponds to the top row.

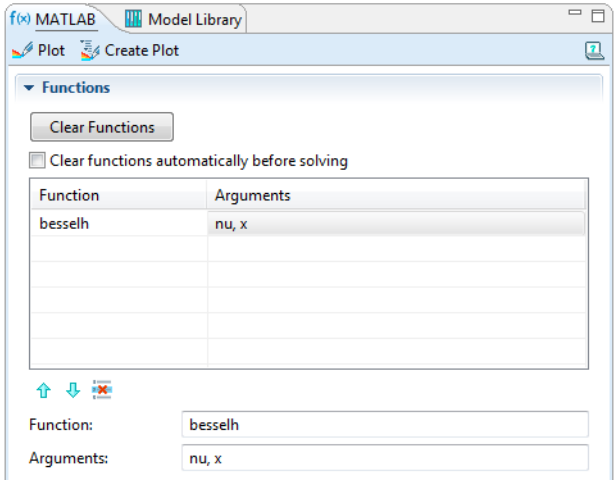
In case there are several functions declared in the **Functions** table, only the function that has the same number of input arguments as the number of filled in rows in the **Plot Parameters** table is plotted.

If several functions have the same number of input arguments, the first function in the table (from top to bottom) is plotted. Use the **Move Up** ( ) and **Move Down** ( ) buttons to change the order of functions in the table.

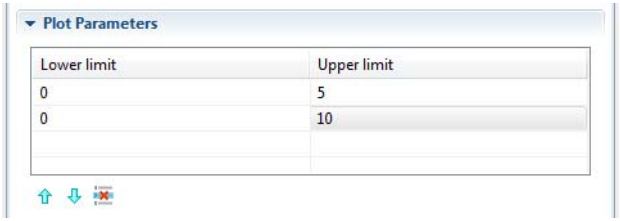
EXAMPLE: DEFINE THE HANKEL FUNCTION

Assume that you want to use MATLAB's Bessel function of the third kind (Hankel function) in a COMSOL model. Add a MATLAB function node, then define the following settings:

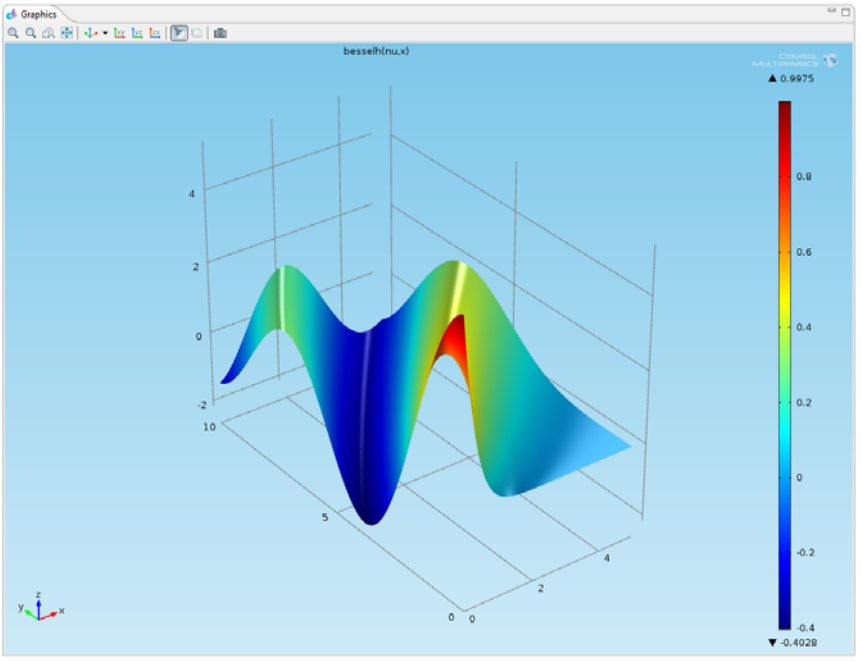
FUNCTION	ARGUMENTS
`besselh`	`nu, x`



To plot the function you need first to define the lower and upper limits for both nu and x. In the **Plot Parameters** table set the first row (which corresponds to the first argument `nu`) of the **Lower limit** column to 0 and the **Upper limit** column to 5 and set the second row (corresponding of `x`) of the **Lower limit** column to 0 and the **Upper limit** column to 10:



the **Plot** button ( ) to get this plot:



Setting the Function Directory Path in MATLAB

To be able to run a model that use an external MATLAB function, the path directory of the function has to be set in MATLAB before it is called by COMSOL to evaluate the function.

To proceed you have three options to set the directory path in MATLAB:

- Save the model MPH-file in the same directory as for the M-functions.

- Set the system environment variable COMSOL_MATLAB_PATH with the M-functions directory path.
- Use the **Set Path** window available in the MATLAB desktop under the **File** menu.

Adding a MATLAB Function with the COMSOL API Java Syntax

To add a MATLAB feature node to the COMSOL model using the COMSOL API, enter the command:

```
model.func.create(<ftag>, 'MATLAB');
```

Define the function name and function arguments with the command:

```
model.func(<ftag>).setIndex('funcs', <function_name>, 0, 0);
model.func(<ftag>).setIndex('funcs', <arglist>, 0, 1);
```

where `<function_name>` is a string set with the function name and `<arglist>` is a string that defines the list of the input arguments.

Function Input/Output Considerations

The functions called from COMSOL must support vector arguments of any length. COMSOL calls a MATLAB function using vector arguments to reduce the number of expensive calls from COMSOL to MATLAB. All common MATLAB functions such as `sin`, `abs`, and other mathematical functions support vector arguments.

When you write your own functions, remember that the input arguments are vectors. The output must have the same size as the input. All arguments and results must be double-precision vectors real or complex valued.

Consider the following example function where the coefficient c depends on the x coordinate:

```
function c = func1(x)
if x > 0.6
  c = x/1.6;
else
  c = x^2+0.3;
end
```

This function looks good at first but it does not work in COMSOL Multiphysics because the input x is a matrix.

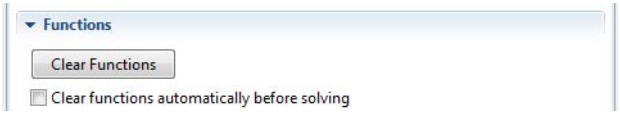
- Element-by-element multiplication, division, and power must be used—that is, the operators `.*`, `./`, and `.^`. Replace expressions such as `x/1.6` and `x^2+0.3` with `x./1.6` and `x.^2+0.3`, respectively.
- The comparison $x > 0.6$ returns a matrix with ones (true) for the entries where the expression holds true and zeros (false) where it is false. The function evaluates the conditional statement if, and only if, all the entries are true (1).

You can replace the `if` statement with a single assignment to the indices retrieved from the $x > 0.6$ operation and another assignment to the indices where $x \leq 0{,}6$. The function could then look like this:

```
function c = func2(x)
c = (x./1.6).*(x>0.6) + (x.^2+0.3).*(x<=0.6);
```

Updating Functions

If the function M-file is modified using a text editor, click **Clear Functions** to ensure the functions modifications is updated in the COMSOL model.



An alternative is to select the **Clear functions automatically before solving** check box.

Defining Function Derivatives

Automatic differentiation is not supported with MATLAB functions. In case the MATLAB function has Jacobian contributions, its derivatives with respect to the function input arguments need to be defined. By default COMSOL assumes the derivatives to be null.

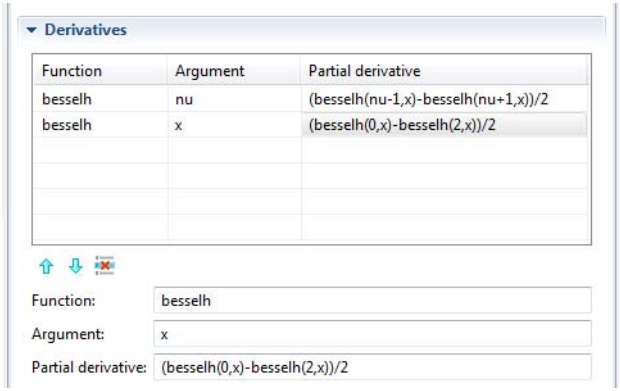
Expand the **Derivatives** section to define the derivatives of the function with respect to the function arguments. In the table define the derivative for each function argument. In the **Function** column enter the function name, in the **Argument** column enter the

argument. Finally in the **Function derivative** column enter the expression for the corresponding derivative.

Note	The function derivatives can also be defined by additional MATLAB functions.

For example, using the Hankel function described in Example: Define the Hankel Function, define the function derivative by entering the following settings in the table:

FUNCTION	ARGUMENT	FUNCTION DERIVATIVE
`besselh`	`nu`	`(besselh(nu-1,x)-besselh(nu+1,x))/2`
`besselh`	`x`	`(besselh(0,x)-besselh(2,x))/2`



Using the MATLAB Debugger (Windows Only)

If you are running on Windows, you can benefit of the MATLAB debugger when running MATLAB functions in the COMSOL model.

To activate the MATLAB debugger, you first need to start the MATLAB Desktop from the MATLAB process that is started when evaluating the function in COMSOL. At the MATLAB prompt enter the command `desktop`.

In the MATLAB Desktop edit the function M-file and add a break point at the desired line. When the COMSOL model is run, the MATLAB debugger automatically stops at the break point. You can then verify the intermediate value of the function variables.

6

Command Reference

The main reference for the syntax of the commands available with LiveLink *for* MATLAB is the *COMSOL Java API Reference Guide*. This section documents additional interface functions that come with the product.

In this chapter:

- Summary of Commands
- Commands Grouped by Function

Summary of Commands

colortable
mphcd
mphdoc
mpheval
mphevalglobalmatrix
mphevalpoint
mphgeom
mphgetadj
mphgetcoords
mphgetexpressions
mphgetproperties
mphgetselection
mphgetu
mphglobal
mphimage2geom
mphinputmatrix
mphinterp
mphload
mphmatrix
mphmax
mphmean
mphmesh
mphmeshstats
mphmin
mphmodel
mphmodellibrary
mphnavigator
mphplot
mphsave
mphsearch
mphselectbox
mphselectcoords
mphshowerrors
mphsolinfo
mphstart

```
mphstate
mphversion
mphviewselection
mphxmeshinfo
```

Commands Grouped by Function

Interface Functions

FUNCTION	PURPOSE
`mphcd`	Change the directory to the directory of the model.
`mphdoc`	Return HTML help of a specified function.
`mphload`	Load a COMSOL model MPH-file.
`mphsave`	Save a COMSOL model.
`mphstart`	Connect MATLAB to a COMSOL server.
`mphversion`	Return the version number of COMSOL Multiphysics.

Geometry Functions

FUNCTION	PURPOSE
`mphgeom`	Plot a geometry in a MATLAB figure.
`mphimage2geom`	Convert image data to geometry.
`mphviewselection`	Display a geometric entity selection in a MATLAB figure.

Mesh Functions

FUNCTION	PURPOSE
`mphmesh`	Plot a mesh in a MATLAB figure.
`mphmeshstats`	Return mesh statistics and mesh data information.

Utility Functions

FUNCTION	PURPOSE
`mphgetadj`	Return geometric entity indices adjacent to each other.
`mphgetcoords`	Return point coordinates of geometry entities.
`mphgetu`	Return solution vectors.
`mphinputmatrix`	Add a linear matrix system to a model.
`mphmatrix`	Get model matrices.
`mphselectbox`	Select a geometric entity using a rubberband/box.

FUNCTION	PURPOSE
mphselectcoords	Select a geometric entity using point coordinates.
mphsolinfo	Get information about a solution object.
mphstate	Get state-space matrices for dynamic systems.
mphxmeshinfo	Extract information about the extended mesh.

Postprocessing Functions

FUNCTION	PURPOSE
mpheval	Evaluate expressions on node points.
mphevalglobalmatrix	Evaluate global matrix variables.
mphevalpoint	Evaluate expressions at geometry vertices.
mphglobal	Evaluate global quantities.
mphinputmatrix	Perform integration of expressions.
mphinterp	Evaluate expressions in arbitrary points or data sets.
mphmax	Perform maximum of expressions.
mphmean	Perform mean of expressions.
mphmin	Perform minimum of expressions.
mphplot	Render a plot group in a figure window.

Model information and navigation

FUNCTION	PURPOSE
mphgetproperties	Get properties from a model node.
mphgetexpressions	Get the model variables and parameters.
mphgetselection	Get information about a selection node.
mphmodel	Return tags for the nodes and subnodes in the COMSOL model object.
mphmodellibrary	GUI for viewing the Model Library.
mphnavigator	GUI for viewing the COMSOL model object.
mphsearch	GUI for searching expressions in the COMSOL model object.
mphshowerrors	Show messages in error and warning nodes in the COMSOL model object.

Purpose

Return a MATLAB colormap for a COMSOL color table.

Syntax

```
map = colortable(name)
```

Description

`map = colortable(name)` returns the color table (of 1024 colors) for `name`, where `name` can be one of the following strings:

`Cyclic` - A color table that varies the hue component of the hue-saturation-value color model, keeping the saturation and value constant (equal to 1). The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. This table is particularly useful for displaying periodic functions and has a sharp color gradient.

`Disco` - This color table spans from red through magenta and cyan to blue.

`Discolight` - Similar to Disco but uses lighter colors.

`Grayscale` - A color table that uses no color, only the gray scale varying linearly from black to white.

`Grayprint` - Varies linearly from dark gray (0.95, 0.95, 0.95) to light gray (0.05, 0.05, 0.05). This color table overcomes two disadvantages that the GrayScale color table has when used for printouts on paper, namely that it gives the impression of being dominated by dark colors, and that white cannot be distinguished from the background.

`Rainbow` - The color ordering in this table corresponds to the wavelengths of the visible part of the electromagnetic spectrum: beginning at the small-wavelength end with dark blue, the colors range through shades of blue, cyan, green, yellow, and red.

`Rainbowlight` - Similar to Rainbow, this color table uses lighter colors.

`Thermal` - Ranges from black through red and yellow to white, which corresponds to the colors iron takes as it heats up.

`Thermalequidistant` - Similar to Thermal but uses equal distances from black to red, yellow, and white, which means that the black and red regions become larger.

`Traffic` - Spans from green through yellow to red.

`Trafficlight` - Similar to Traffic but uses lighter colors.

`Wave` - Ranges linearly from blue to light gray, and then linearly from white to red. When the range of the visualized quantity is symmetric around zero, the color red

or blue indicates whether the value is positive or negative, and the saturation indicates the magnitude.

`Wavelight` - Similar to Wave and ranges linearly from a lighter blue to white (instead of light gray) and then linearly from white to a lighter red.

Calling `colortable` is equivalent to calling the corresponding colormap function directly.

Example

Create a rainbow color map

```
map = colortable('Rainbow');
map = rainbow;
```

Purpose Change directory to the directory of the model

Syntax `mphcd(model)`

Description `mphcd(model)` changes the current directory in MATLAB to the directory where the `model` was last saved.

See aalso `mphload`, `mphsave`

Purpose

Return HTML help of a specified function.

Syntax

```
mphdoc arg1
mphdoc arg1 arg2
```

Description

`mphdoc arg1` returns the HTML documentation associated to the function `arg1`.

`mphdoc arg1 arg2` returns the HTML documentation associated to the feature `arg2` of the method `arg1`.

`mphdoc arg1 -web` returns the HTML documentation in the default web browser.

Example

Create a model object

```
model = ModelUtil.creat('Model')
```

Get the documentation for the mesh node

```
mphdoc model.mesh
```

Get the documentation of the rectangle geometry feature

```
mphdoc model.geom Rectangle
```

Display the documentation in the default web browser

```
mphdoc model.sol -web
```

Purpose

Evaluate expressions on node points.

Syntax

```
pd = mpheval(model,{e1,...,en},...)
```

Description

`pd = mpheval(model,{e1,...,en},...)` returns the post data `pd` for the expressions `e1,...,en`.

The output value `pd` is a structure with fields `expr`, `p`, `t`, `ve`, `unit` and fields for data values.

- The field `expr` contains the expression name evaluated.
- For each expression `e1,...,en` a field with the name `d1,... dn` is added with the numerical values. The columns in the data value fields correspond to node point coordinates in columns in `p`. The data contains only the real part of complex-valued expressions.
- The field `p` contains node point coordinate information. The number of rows in `p` is the number of space dimensions.
- The field `t` contains the indices to columns in `p` of a simplex mesh, each column in `t` representing a simplex.
- The field `ve` contains indices to mesh elements for each node point.
- The field `unit` contains the list of the unit for each expression.

The function `mpheval` accepts the following property/value pairs:

TABLE 6-1: PROPERTY/VALUE PAIRS FOR THE MPHEVAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Complexfun`	`off | on`	`on`	Use complex-valued functions with real input
`Complexout`	`off | on`	`off`	Return complex values
`Dataonly`	`off | on`	`off`	Only return expressions value
`Dataset`	`String`		Data set tag
`Edim`	`point | edge | boundary | domain | 0 | 1 | 2 | 3`	`Geometry space dimension`	Evaluate on elements with this space dimension
`Matherr`	`off | on`	`off`	Error for undefined operations or expressions
`Outersolnum`	`Positive integer`	`1`	Solution number for parametric sweep

TABLE 6-1: PROPERTY/VALUE PAIRS FOR THE MPHEVAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
Pattern	lagrange | gauss	lagrange	Specifies if evaluation takes place in Lagrange points or in Gauss points
Phase	Scalar	0	Phase angle in degrees
Recover	off | ppr | pprint	off	Accurate derivative recovery
Refine	Integer	1	Refinement of elements for evaluation points
Selection	Integer vector | string | all	All domains	Set selection tag or entity number
Smooth	Internal | none | everywhere	internal	Smoothing setting
Solnum	Integer vector | all | end	all	Solutions for evaluation
t	Double array		Times for evaluation

The property `Dataset` controls which data set is used for the evaluation. Data Sets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution Data Sets.

The property `Edim` decides which elements to evaluate on. Evaluation takes place only on elements with space dimension `Edim`. If not specified, `Edim` equal to the space dimension of the geometry is used. The setting is specified as one of the following strings `'point'`, `'edge'`, `'boundary'` or `'domain'`. In previous versions it was only possible to specify `Edim` as a number. For example, in a 3D model, if evaluation is done on edges (1D elements), `Edim` is 1. Similarly, for boundary evaluation (2D elements), `Edim` is 2, and for domain evaluation (3D elements), Edim is 3 (default in 3D).

Use `Recover` to recover fields using polynomial-preserving recovery. This techniques recover fields with derivatives such as stresses or fluxes with a higher theoretical convergence than smoothing. Recovery is expensive so it is turned off by default. The value `pprint` means that recovery is performed inside domains. The value `ppr` means that recovery is also applied on all domain boundaries.

The property `Refine` constructs evaluation points by making a regular refinements of each element. Each mesh edge is divided into `Refine` equal parts.

The property `Smooth` controls if the post data is forced to be continuous on element edges. When `Smooth` is set to internal, only elements not on interior boundaries are made continuous.

The property `Solnum` is used to select the solution to plot when a parametric, eigenvalue or time dependent solver has been used to solve the problem.

The property `Outersolnum` is used to select the solution to plot when a parametric sweep has been used in the study.

When the property `Phase` is used, the solution vector is multiplied with `exp(i*phase)` before evaluating the expression.

The expressions `e1,...,en` are evaluated for one or several solutions. Each solution generates an additional row in the data fields of the post data output structure. The property `Solnum` and `t` control which solutions are used for the evaluations. The `Solnum` property is available when the data set has multiple solutions, for example in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

For time-dependent problems, the variable `t` can be used in the expressions `ei`. The value of `t` is the interpolation time when the property `t` is provided, and the time for the solution, when `Solnum` is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

Example

Load `micromixer.mph` from the Model Library:

```
model = mphload('micromixer.mph');
```

Evaluate the pressure `p` at all node points:

```
dat = mpheval(model,'p');
```

Evaluate the concentration `c` at the outlet boundary:

```
dat = mpheval(model,'c','edim','boundary','selection',136);
```

Evaluate the pressure on all geometric vertices:

```
dat = mpheval(model,'p','edim','point');
```

Evaluate the pressure on vertex number 1 and return only the pressure value:

```
dat = mpheval(model,'p','edim','point',...
'selection',1,'dataonly','on');
```

See also mphevalpoint, mphglobal, mphinputmatrix, mphinterp

Purpose

Evaluate global matrix variables.

Syntax

```
M = mphevalglobalmatrix(model,expr,...)
```

Description

`M = mphevalglobalmatrix(model,expr,...)` evaluates the global matrix of the variable `expr` and returns the full matrix `M`.

The function `mphevalglobalmatrix` accepts the following property/value pairs:

TABLE 6-2: PROPERTY/VALUE PAIRS FOR THE MPHEVAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Dataset`	`String`		Data set tag

Note: S-parameters evaluation requires the RF module.

Example

Load `lossy_circulator_3d.mph` from the RF Module's Model Library:

```
model = mphload('lossy_circulator_3d.mph');
```

Evaluate the S-parameters matrix using the solution data set `dset4`:

```
M = mphevalglobalmatrix(model,'emw.SdB','dataset','dset4');
```

See also

`mpheval`, `mphinterp`, `mphglobal`

Purpose

Evaluate expressions at geometry vertices.

Syntax

```
[v1,...,vn] = mphevalpoint(model,{e1,...,en},...)
[v1,...,vn,unit] = mphevalpoint(model,{e1,...,en},...)
```

Description

`[v1,...,vn] = mphevalpoint(model,{e1,...,en},...)` returns the results from evaluating the expressions `e1,...,en` at the geometry vertices. The values `v1,...,vn` can either be a cell array or a matrix depending on the options.

`[v1,...,vn,unit] = mphevalpoint(model,{e1,...,en},...)` also returns the unit of all expressions `e1,...,en` in the 1xN cell array `unit`.

The function `mphevalpoint` accepts the following property/value pairs:

TABLE 6-3: PROPERTY/VALUE PAIRS FOR THE MPHEVAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Dataset`	`String`		Data set tag
`Dataseries`	`none | mean | int | max | min | rms | std | var`	`none`	The operation that is applied to the data series formed by the evaluation
`Matrix`	`off | on`	`on`	Return a matrix if possible
`Minmaxobj`	`Real | abs`	`real`	The value being treated if `Dataseries` is set to `max` or `min`
`Outersolnum`	`Positive integer`	`1`	Solution number for parametric sweep
`Selection`	`Integer vector | string | all`	`All domains`	Set selection tag or entity number
`Smooth`	`Internal | none | everywhere`	`internal`	Smoothing setting
`Solnum`	`Integer vector | all | end`	`all`	Solutions for evaluation
`Squeeze`	`on | off`	`on`	Squeeze singleton dimension
`t`	`Double array`		Times for evaluation

The property `Dataset` controls which data set is used for the evaluation. Data Sets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution Data Sets.

The `Dataseries` property is used to control any filtering of the data series. The supported operations are: average (`mean`), integral (`int`), maximum (`max`), minimum (`min`), root mean square (`rms`), standard deviation (`std`) and variance (`var`).

Set the property `Matrix` to `off` to get the results in a cell array format.

In case the property `Datseries` is either `min` or `max`, you can specify the how the value are treated using the property `Minmaxobj`. Use either the real data or the absolute data.

The property `Solnum` is used to select the solution to plot when a parametric, eigenvalue or time dependent solver has been used to solve the problem.

The expressions `e1,...,en` are evaluated for one or several solutions. Each solution generates an additional row in the data fields of the post data output structure. The property `Solnum` and `t` control which solutions are used for the evaluations. The `Solnum` property is available when the data set has multiple solutions, for example in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

For time-dependent problems, the variable `t` can be used in the expressions `ei`. The value of `t` is the interpolation time when the property `t` is provided, and the time for the solution, when `Solnum` is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

Example

Load `shallow_water_equations.mph` from the Model Library:

```
model = mphload('shallow_water_equations.mph');
```

Evaluate the elevation `Z` at point number 2:

```
dat = mphevalpoint(model,'Z','selection',2);
```

Evaluate the maximum value of the elevation with respect to the time at point 2:

```
dat = mphevalpoint(model,'Z','selection',2,'dataseries','max');
```

Evaluate the maximum value of the elevation with respect to the time at point 2:

```
dat = mphevalpoint(model,'Z','selection',2,'dataseries','rms'));
```

See also mpheval, mphglobal, mphinputmatrix, mphinterp

Purpose

Plot a geometry in a MATLAB figure.

Syntax

```
mphgeom(model)
mphgeom(model,geomtag,...)
```

Description

`mphgeom(model)` plots the model geometry in a MATLAB figure.

`mphgeom(model,geomtag,...)` plots the model geometry with the tag `geomtag` in a MATLAB figure.

The function `mphgeom` accepts the following property/value pairs:

TABLE 6-4: PROPERTY/VALUE PAIRS FOR THE MPHGEOM COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Parent`	`Double`		Parent axes
`Selection`	`Positive integer array`		Selection
`Entity`	`point | edge | boundary | domain`		Geometric entity to select
`Build`	`on | off | current | string`	`on`	Build the geometry before plotting
`Edgecolor`	`Char`	`k`	Edge color
`Edgelabels`	`on | off`	`off`	Show edge labels
`Edgelabelscolor`	`Char`	`k`	Color for edge labels
`Edgemode`	`on | off`	`on`	Show edges
`Facealpha`	`Double`	`1`	Set transparency value
`Facelabels`	`on | off`	`off`	Show face labels
`Facelabelscolor`	`Char`	`k`	Color for face labels
`Facemode`	`on | off`	`on`	Show faces
`Vertexlabels`	`on | off`	`off`	Show vertex labels
`Vertexlabelscolor`	`Char`	`k`	Color for vertex labels
`Vertexmode`	`on | off`	`off`	Show vertices

The `Build` property determines if `mphgeom` build the geometry prior to display it. If the `Build` property is set with a geometric object tag, the geometry is built up to that object. `mphgeom` only displays built geometry objects.

Example

Load `shell_diffusion.mph` from the Model Library:

```
model = mphload('shell_diffusion.mph');
```

Plot the geometry:

```
mphgeom(model)
```

Plot the geometry and boundaries 2,4,6,8,10,16,17,18,29 and 20 in subplot:

```
ax = subplot(1,2,1);
mphgeom(model,'geom1', ...
    'parent',ax, ...
    'edgecolor','k', ...
    'edgelabels','on', ...
    'alpha',0.5,...
    'edgelabelscolor','b', ...
    'vertexmode','on',...
    'edgemode','on');
ax = subplot(1,2,2);
mphgeom(model,'geom1', ...
    'parent',ax, ...
    'entity','boundary', ...
    'selection',[2:2:10,16:20]);
```

See also `mphmesh`, `mphviewselection`

Purpose

Return geometry entity indices that are adjacent to other.

Syntax

```
n = mphgetadj(model,geomtag,returntype,adjtype,adjnumber)
```

Description

`n = mphgetadj(model,geomtag,returntype,adjtype,adjnumber)` returns the indices of the adjacent geometry entities.

`returntype` is the type of the geometry entities whose index are returned.

`adjtype` is the type of the input geometry entity.

The entity type can be one of `'point'`, `'edge'`, `'boundary'` or `'domain'` following the entity space dimension defined below:

- `'domain'`: maximum geometry space dimension
- `'boundary'`: maximum geometry space dimension -1
- `'edges'`: 1(only for 3D geometry)
- `'point'`: 0

Example

Load `busbar.mph` from the Model Library:

```
model = mphload('busbar.mph');
```

Returns the indices of the domains adjacent to point 2:

```
n = mphgetadj(model,'geom1','domain','point',2);
```

Returns the indices of the points adjacent to domains 2 and 3:

```
n = mphgetadj(model,'geom1','point','domain',[2 3]);
```

See also

`mphgetcoords`, `mphselectbox`, `mphselectcoords`

Purpose

Return point coordinates of geometry entities.

Syntax

```
c = mphgetcoords(model,geomtag,entitytype,entitynumber)
```

Description

`c = mphgetcoords(model,geomtag,entitytype,entitynumber)` returns the coordinates of the points that belong to the entity object with the type `entitytype` and the index `entitynumber`.

The `entitytype` property can be one of `'point'`, `'edge'`, `'boundary'` or `'domain'` following the entity space dimension defined below:

- `'domain'`: maximum geometry space dimension
- `'boundary'`: maximum geometry space dimension -1
- `'edge'`: 1 (only for 3D geometry)

`'point'`: 0

Example

Load `busbar.mph` from the Model Library:

```
model = mphload('busbar.mph');
```

Return the coordinates of points that belong to domain 1:

```
c = mphgetcoords(model,'geom1','domain',1);
```

Return the coordinates of points that belong to boundary 5:

```
c = mphgeomcoords(model,'geom1','boundary',5);
```

Return the coordinates of point number 10:

```
c = mphgeomcoords(model,'geom1','point',10);
```

See also

`mphgetadj`, `mphselectbox`, `mphselectcoords`

Purpose Get the model variables and model parameters expressions.

Syntax

```
expr = mphgetexpressions(modelnode)
```

Description `expr = mphgetexpressions(modelnode)` returns expressions from the node `modelnode` as a cell array. `expr` contains the list of the variable names, the variable expressions and the variable descriptions.

Note that not all nodes have expressions defined.

Example Load example model `busbar.mph` from the Model Library:

```
model = mphload('stresses_in_pulley.mph');
```

Get the expressions defined in the parameters node:

```
expr = mphgetexpressions(model.param)
```

See also `mphnavigator`, `mphmodel`

Purpose Get the properties from a model node

Syntax `mphproperties(modelnode)`

Description `mphproperties(modelnode)` returns properties that are defined for the node `modelnode`.

Example Load busbar.mph from the Model Library:

```
model = mphload('busbar');
```

Get the properties of the node `model.result('pg1')`:

```
prop = mphgetproperties(model.result('pg1'))
```

See also `mphnavigator`

Purpose

Get information about a selection node.

Syntax

```
info = mphgetselection(selnode)
```

Description

`info = mphgetselection(selnode)` returns the selection data of the selection node `selnode`.

The output `info` is a MATLAB structure defined with the following fields:

- `dimension`, the space dimension of the geometry entity selected.
- `geom`, the geometry tag.
- `entities`, the indexes of the selected entities.
- `isGlobal`, a Boolean expression that indicates if the selection is global.

Example

Load `busbar.mph` from the Model Library:

```
model = mphload('busbar.mph');
```

Get the information of the selection node `model.selection('sel1')`:

```
info = mphgetselection(model.selection('sel1'))
```

See also

`mphnavigator`

Purpose

Return solution vector.

Syntax

```
U = mphgetu(model,...)
[U,Udot] = mphgetu(model,...)
```

Description

`U = mphgetu(model)` returns the solution vector `U` for the default solution data set.

`[U,Udot] = mphgetu(model,...)` returns in addition `Udot`, which is the time derivative of the solution vector. This syntax is available for a time-dependent solution only.

For a time-dependent and parametric analysis type, the last solution is returned by default. For an eigenvalue analysis type the first solution number is returned by default.

The function `mphgetu` accepts the following property/value pairs:

TABLE 6-5: PROPERTY/VALUE PAIRS FOR THE MPHGETU COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Solname`	`String`	`Auto`	Solver node tag
`Solnum`	`Positive integer vector`	`Auto`	Solution for evaluation
`Type`	`String`	`Sol`	Solution type
`Matrix`	`off | on`	`on`	Store as matrix if possible

The `Solname` property set the solution data set to use associated with the defined solver node.

Type is used to select the solution type. This is `'Sol'` by default. The valid types are: `'Sol'` (main solution), `'Reacf'` (reaction force), `'Adj'` (adjoint solution), `'Fsens'` (functional sensitivity) and `'Sens'` (forward sensitivity).

If `Solnum` is a vector and the result has been obtained with the same mesh then the solution will be stored in a matrix if the `Matrix` option is set to `'on'`

Example

Load `falling_sand.mph` from the Model Library:

```
model = mphload('falling_sand.mph');
```

Get the solution vector for the last time step (default solution number):

```
u = mphgetu(model);
```

Get the solution vector and its derivative for solution number 9 and 10:

```
[u,ut] = mphgetu(model,'solnum',[9 10]);
```

See also `mphsolinfo`

Purpose

Evaluate global quantities.

Syntax

```
[d1,...,dn] = mphglobal(model,{e1,...,en},...)
[d1,...,dn,unit] = mphglobal(model,{e1,...,en},...)
```

Description

`[d1,...,dn] = mphglobal(model,{e1,...,en},...)` returns the results from evaluating the global quantities specified in the string expression `e1,..., en`.

`[d1,...,dn,unit] = mphglobal(model,{e1,...,en},...)` also returns the unit of the expressions `e1,..., en`. `unit` is a `nx1` cell array.

The function `mphglobal` accepts the following property/value pairs:

TABLE 6-6: PROPERTY/VALUE PAIRS FOR THE MPHGLOBAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Complexfun`	`off | on`	`on`	Use complex-valued functions with real input
`Complexout`	`off | on`	`off`	Return complex values
`Dataset`	`String`	Active solution data set	Data set tag
`Matherr`	`off | on`	`off`	Error for undefined operations or expressions
`Outersolnum`	`Positive integer`	`1`	Solution number for parametric sweep
`Phase`	`Scalar`	`0`	Phase angle in degrees
`Solnum`	`Integer vector | all | end`	`all`	Solution for evaluation
`T`	`Double array`		Time for evaluation
`Unit`	`String | cell array`		Unit to use for the evaluation

The property `Dataset` controls which data set is used for the evaluation. Data Sets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on solution data sets.

When the property `Phase` is used, the solution vector is multiplied with `exp(i*phase)` before evaluating the expression.

The expressions `ei` are evaluated for one or several solutions. Each solution generates an additional row in the output data array `di`. The property `Solnum` and `t` control which solutions are used for the evaluations. The `Solnum` property is available when the data set has multiple solutions, for example in the case of

parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

For time-dependent problems, the variable `t` can be used in the expressions `ei`. The value of `t` is the interpolation time when the property `t` is provided, and the time for the solution, when `Solnum` is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

In case of multiple expression if the `unit` property is defined with a string, the same unit is used for both expressions. To use different units, set the property with a cell array. In case of inconsistent unit definition, the default unit is used instead.

Solnum is used to select the solution number when a parametric, eigenvalue or time-dependent solver has been used.

`Outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

Example

Load `fluid_valve.mph` from the Model Library:

```
model = mphload('fluid_valve.mph');
```

Evaluate the global expression `u_up` for each time step:

```
u_up = mphglobal(model,'u_up')
```

Evaluate the global expression `u_up` at $t = 0.8$ sec:

```
u_up = mphglobal(model,'u_up','t',0.8)
```

Evaluate the expressions `u_up` and `u_down` at the last solution number:

```
[u_up,u_down] =
mphglobal(model,{'u_up','u_down'},'solnum','end')
```

Evaluate the expressions `u_up` in mm^2/s and `u_down` in cm^2/s:

```
[u_up,u_down] =
mphglobal(model,{'u_up','u_down'},'unit',{'mm^2/s','cm^2/s'});
```

See also

`mpheval`, `mphevalpoint`, `mphinputmatrix`, `mphinterp`

Purpose

Convert image data to geometry.

Syntax

```
model = mphimage2geom(imagedata,level,...)
```

Description

`model = mphimage2geom(imagedata,level,...)` converts the image contained in imagedata into a geometry which is returned in the model object model.

The contour of the image is defined by the value level. `imagedata` must be a 2D matrix.

The function `mphimage2geom` accepts the following property/value pairs:

TABLE 6-7: PROPERTY/VALUE PAIRS FOR THE MPHIMAGE2GEOM COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Rtol`	`Value`	`1e-3`	Relative tolerance for interpolation curves
`Type`	`Solid | closed | open`	`solid`	Type of geometry object
`Curvetype`	`Auto | polygon`	auto	Type of curve to create the geometry object
`Scale`	`Value`	`1`	Scale factor from pixels to geometry scale
`Mindist`	`Value`	`1`	Minimum distance between coordinates in curves (in pixels)
`Compose`	`on | off`	`on`	Create compose nodes for overlapping solids
`Rectangle`	`on | off`	`off`	Insert rectangle in the geometry

The default curve types creates a geometry with the best suited geometrical primitives. For interior curves this is Interpolation Curves and for curves that are touching the perimeter of the image, Polygons is used.

Example

Create the geometry following the contour level 50 of an image data array provided by the function peaks:

```
p = (peaks+7)*5;
figure(1)
[c,h] = contourf(p)
clabel(c,h);
colorbar
model = mphimage2geom(p, 50);
figure(2);
```

```
mphgeom(model)
```

Purpose

Add linear matrix system to a model.

Syntax

```
mphinputmatrix(model,str,soltag,soltypetag)
```

Description

`mphinputmatrix(model,str,soltag,soltypetag)` adds the linear system matrices and vectors stored in the MATLAB structure `str` to the `model`. The system matrices is associated to the solver configuration defined with the tag `soltag` and solved with the solver defined with the tag `soltypetag`.

`soltypetag` can only be one of the following solver type: Stationary, Eigenvalue, Time.

A valid structure for a stationary solver includes the following fields:

TABLE 6-8: PROPERTY/VALUE PAIRS FOR THE MPHINT2 COMMAND.

FIELD NAME	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian

A valid structure for a time dependent/ eigenvalue solver includes the following fields:

TABLE 6-9: PROPERTY/VALUE PAIRS FOR THE MPHINT2 COMMAND.

FIELD NAME	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian
D	Damping matrix
E	Mass matrix

There is also the possibility to include the constraint force Jacobian vector `NF`.

Once the matrix system is loaded in the model, the solver configuration is set ready to run.

Note: The system matrices are not stored in the model when it is saved as a MPH-file or loaded into the COMSOL Desktop.

Example

Load the model `model_tutorial_llmatlab.mph`:

```
model = mphload('model_tutorial_llmatlab.mph');
```

Add a stationary study step and a solver configuration:

```
std1 = model.study.create('std1');
std1.feature.create('stat','Stationary');
sol1 = model.sol.create('sol1');
sol1.study('std1');
```

Extract the linear stationary matrix system in MATLAB:

```
str = mphmatrix(model,'sol1','out',{'K','L','M','N',},...
'initmethod','sol','initsol','zero');
```

Change the linear system by scaling the stiffness matrix:

```
str.K = str.K*0,1;
```

Set-up the solver configuration for a stationary problem:

```
st1 = sol1.feature.create('st1','StudyStep');
st1.set('studystep','stat');
sol1.feature.create('s1','Stationary');
```

Insert the system matrix back to the model:

```
mphinputmatrix(model,str,'sol1','s1');
```

Run the solver configuration:

```
model.sol('sol1').runAll;
```

See also

`mphmatrix`, `mphxmeshinfo`

Purpose

Perform integration of expressions.

Syntax

```
[v1,...,v2] = mphint2(model,{e1,...,en},edim,...)
[v1,...,v2,unit] = mphint2(model,{e1,...,en},edim,...)
```

Description

`[v1,...,vn] = mphint2(model,{e1,...,en},...)` evaluates the integrals of the string expressions `e1,...,en` and returns the result in N matrices `v1,...,vn` with M rows and P columns. M is the number of inner solution and P the number of outer solution used for the evaluation. `edim` defines the element dimension, as a string: `line`, `surface`, `volume` or as an integer value.

`[v1,...,vn] = mphint2(model,{e1,...,en},...)` also returns the units of the integral in a 1xN cell array.

The function `mphint2` accepts the following property/value pairs:

TABLE 6-10: PROPERTY/VALUE PAIRS FOR THE MPHINT2 COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Dataset`	`String`	`active solution data set`	Data set tag
`Intorder`	`Positive integer`	`4`	Integration order
`Intsurface`	`on | off`	`off`	Compute surface integral
`Intvolume`	`on | off`	`off`	Compute volume integral
`Matrix`	`off | on`	`on`	Returns data as a matrix or as a cell
`Method`	`auto | integration | summation`	`auto`	Integration method
`Outersolnum`	`Positive integer`	`1`	Solution number for parametric sweep
`Selection`	`Integer vector | string | all`	`all`	Selection list or named selection
`Solnum`	`Integer vector | end | all`	`all`	Solution for evaluation
`Squeeze`	`on | off`	`on`	Squeeze singleton dimensions
`T`	`Double array`		Time for evaluation

The property `Dataset` controls which data set is used for the evaluation. Data Sets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution Data Sets.

The expressions `e1,...,en` are integrated for one or several solutions. Each solution generates an additional column in the returned matrix. The property `Solnum` and `t` control which solutions are used for the integrations. The Solnum property is available when the data set has multiple solutions, for example in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

For time-dependent problems, the variable `t` can be used in the expressions `ei`. The value of `t` is the interpolation time when the property `t` is provided, and the time for the solution, when `Solnum` is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

The `unit` property defines the unit of the integral, if a inconsistent unit is entered, the default unit is used. In case of multiple expression, if the `unit` property is defined with a string, the same unit is used for both expressions. To use different units, set the property with a cell array. In case of inconsistent unit definition, the default unit is used instead.

Solnum is used to select the solution number when a parametric, eigenvalue or time-dependent solver has been used.

`Outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

Example

Load `micromixer.mph` from the Model Library:

```
model = mphload('micromixer.mph');
```

Integrate the *x*-velocity `u` at the outlet boundary and get its unit:

```
[flow unit]= mphint2(model,'u','surface','selection',136)
```

Load `heat_transfer_axi.mph` from the Model Library:

```
model = mphload('heat_transient_axi.mph');
```

Integrate the normal heat flux along the external boundaries using surface integration:

```
Q = mphint2(model,'ht.ndflux',1,'intsurface','on',...
```

```
  'selection',[2,3,4]);
```

See also mpheval, mphevalpoint, mphglobal, mphinterp

Purpose

Evaluate expressions in arbitrary points or data sets.

Syntax

```
[v1,...,vn] = mphinterp(model,{e1,...,en},'coord',coord,...)
[v1,...,vn] = mphinterp(model,{e1,...,en},'dataset',dsettag,...)
[v1,...,vn,unit] = mphinterp(model,{e1,...,en},...)
```

Description

`[v1,...,vn] = mphinterp(model,{e1,...,en},'coord',coord,...)` evaluates expressions `e1,...en` at the coordinates specified in the double matrix `coord`. Evaluation is supported only on Solution Data Sets.

`[v1,...,vn] = mphinterp(model,{e1,...,en},'dataset',dsettag,...)` evaluates expressions `e1,...en` on the specified data set `dsettag`. In this case the data set needs to be of a type that defines an interpolation in itself, such as cut planes, revolve, and so forth.

`[v1,...,vn,unit] = mphinterp(model,{e1,...,en},...)` returns in addition the unit of the expressions.

The function `mphinterp` accepts the following property/value pairs:

TABLE 6-11: PROPERTY/VALUE PAIRS FOR THE MPHINTERP COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Complexfun`	`off | on`	`on`	Use complex-valued functions with real input
`Complexout`	`off | on`	`off`	Return complex values
`Coord`	`Double array`		Coordinates for evaluation
`Coorderr`	`off | on`	`off`	Give an error message if all coordinates are outside the geometry
`Dataset`	`String`	`Auto`	Data set tag
`Edim`	`'point' | 'edge' | 'boundary' | 'domain' | 0 | 1 | 2 | 3`	`Geometry space dimension`	Element dimension for evaluation
`Ext`	`Value`	`0.1`	Extrapolation control
`Matherr`	`off | on`	`off`	Error for undefined operations or expressions
`Outersolnum`	`Positive integer`	`1`	Solution number for parametric sweep

TABLE 6-11: PROPERTY/VALUE PAIRS FOR THE MPHINTERP COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
Phase	Scalar	0	Phase angle in degrees
Recover	off | ppr | pprint	off	Accurate derivative recovery
Selection	Positive Integer array | all	all	Selection list
Solnum	Positive integer array | all | end	all	Inner solutions for evaluation
T	Double array		Time for evaluation
Unit	String | Cell array		Unit to use for the evaluation

The columns of the matrix `coord` are the coordinates for the evaluation points. If the number of rows in `coord` equals the space dimension, then `coord` are global coordinates, and the property `Edim` determines the dimension in which the expressions are evaluated. For instance, `Edim='boundary'` means that the expressions are evaluated on boundaries in a 3D model. If `Edim` is less than the space dimension, then the points in `coord` are projected onto the closest point on a domain of dimension `Edim`. If, in addition, the property `Selection` is given, then the closest point on domain number `Selection` in dimension `Edim` is used.

If the number of rows in `coord` is less than the space dimension, then these coordinates are parameter values on a geometry face or edge. In that case, the domain number for that face or edge must be specified with the property `Selection`.

The expressions that are evaluated can be expressions involving variables, in particular physics interface variables.

The matrices `v1,...,vn` are of the size `k`-by-size(`coord`,2), where `k` is the number of solutions for which the evaluation is carried out, see below. The value of expression `ei` for solution number `j` in evaluation point `coord(:,m)` is `vi(j,m)`.

The vector `pe` contains the indices `m` for the evaluation points `code(:,m)` that are outside the mesh, or, if a domain is specified, are outside that domain.

The property `Data` controls which data set is used for the evaluation. Data Sets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution Data Sets. The active solution data set is used by default.

The property `Edim` decides which elements to evaluate on. Evaluation takes place only on elements with space dimension `Edim`. If not specified, `Edim` equal to the space dimension of the geometry is used. The setting is specified as one of the following strings `'point', 'edge', 'boundary'` or `'domain'`. In previous versions it was only possible to specify `Edim` as a number. For example, in a 3D model, if evaluation is done on edges (1D elements), `Edim` is 1. Similarly, for boundary evaluation (2D elements), `Edim` is 2, and for domain evaluation (3D elements), Edim is 3 (default in 3D).

Use `Recover` to recover fields using polynomial-preserving recovery. This techniques recover fields with derivatives such as stresses or fluxes with a higher theoretical convergence than smoothing. Recovery is expensive so it is turned off by default. The value `pprint` means that recovery is performed inside domains. The value `ppr` means that recovery is also applied on all domain boundaries.

The property `Refine` constructs evaluation points by making a regular refinements of each element. Each mesh edge is divided into `Refine` equal parts.

The property `Smooth` controls if the post data is forced to be continuous on element edges. When `Smooth` is set to internal, only elements not on interior boundaries are made continuous.

When the property `Phase` is used, the solution vector is multiplied with `exp(i*phase)` before evaluating the expression.

The expressions `e1,...,en` are evaluated for one or several solutions. Each solution generates an additional row in the data fields of the post data output structure. The property `Solnum` and `t` control which solutions are used for the evaluations. The `Solnum` property is available when the data set has multiple solutions, for example, in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

For time-dependent problems, the variable `t` can be used in the expressions `ei`. The value of `t` is the interpolation time when the property `t` is provided, and the time for the solution, when `Solnum` is used. Similarly, `lambda` and the parameter are

available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

In case of multiple expression, if the `unit` property is defined with a string, the same unit is used for both expressions. To use different units, set the property with a cell array. In case of inconsistent unit definition, the default unit is used instead.

`Solnum` is used to select the solution number when a parametric, eigenvalue or time-dependent solver has been used.

`Outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

Example

Load `heat_convection_2d.mph` from the Model Library:

```
model = mphload('heat_convection_2d.mph');
```

Compute the temperature `T` at the center of the domain:

```
T = mphinterp(model,'T','coord',[0.3;0.5])
```

Load `transport_and_adsorption.mph` from the Model Library:

```
model = mphload('transport_and_asorption.mph');
```

Evaluate the concentration along boundary 5 at $t = 2$ s:

```
list = [0:1e-3:0.1 0.2:0.1:0.9 0.9:1e-3:1];
[c,y] = mphinterp(model,{'c','y'},'coord',list,'edim',1,...
                'selection',5,'t',2);
```

Load `stresses_in_pulley.mph` from the Model Library:

```
model = mphload('stresses_in_pulley.mph');
```

Evaluate the von Mises effective stress at the cut point data set cpt1:

```
[mises,n]= mphinterp(model,{'solid.mises','n'},...
    'datset','cpt1');
```

See also

`mpheval`, `mphevalpoint`, `mphglobal`, `mphinputmatrix`

Purpose

Load a COMSOL model MPH-file.

Syntax

```
model = mphload(filename)
model = mphload(filename, ModelObjectName)
model = mphload(filename, ModelObjectName, '-history')
[model, filename] = mphload(filename, ModelObjectName)
```

Description

`model = mphload(filename)` loads a COMSOL model object saved with the name `filename` and assigns the default name `Model` in the COMSOL server.

`model = mphload(filename, ModelObjectName)` loads a COMSOL model object and assigns the name `ModelObjectName` in the COMSOL server.

`model = mphload(filename, ModelObjectName, '-history')` turns on history recording.

`[model, filenameloaded] = mphload(filename, ModelObjectName)` also returns the full file name `filenameloaded` of the file that was loaded.

If the model name is the same as a model that is currently in the COMSOL server the loaded model overwrites the existing one.

Note that MATLAB searches for the model on the MATLAB path if an absolute path is not supplied.

`mphload` turns off the model history recording by default, unless the property `'-history'` is used.

The extension `mph` can be omitted.

Example

Load `transport_and_adsorption.mph` from the Model Library:

```
model = mphload('transport_and_asorption.mph');
```

Load `stresses_in_pulley.mph` without specifying the mph extension:

```
model = mphload('stresses_in_pulley');
```

Load the model from `MyModel.mph` with the path specified:

```
model = mphload('PATH\MyModel.mph');
```

Load `effective_diffusivity.mph` from the Model Library and return the file name:

```
[model, filename] = mphload('effective_diffusivity.mph');
```

See also

`mphsave`

Purpose

Get model matrices.

Syntax

```
str = mphmatrix(model,soltag,'Out',...)
```

Description

`str = mphmatrix(model,soltag,'Out',{'A'},...)` returns a MATLAB structure `str` containing the matrix `A` assembled using the solver node `soltag` and accessible as `str.A`. `A` being taken from the `Out` property list.

`str = mphmatrix(model,soltag,fname,'Out',{'A','B',...})` returns a MATLAB structure `str` containing the matrices `A`, `B`, ... assembled using the solver node `solname` and accessible as `str.A` and `str.B`. `A` and `B` being taken from the `Out` property list.

The function `mphmatrix` accepts the following property/value pairs:

TABLE 6-12: PROPERTY/VALUE PAIRS FOR THE MPHMATRIX COMMAND

PROPERTY	EXPRESSION	DEFAULT	DESCRIPTION
`out`	`Cell array of strings`		List of matrices to assemble
`Initmethod`	`init | sol`	`sol`	Use linearization point
`Initsol`	`string | zero`	`Active solver tag`	Solution to use for linearization
`Solnum`	`Positive integer| auto`	`auto`	Solution number

The following values are valid for the `out` property:

Property/Value Pairs for the property out.

PROPERTY	EXPRESSION	DESCRIPTION
`out`	`K`	Stiffness matrix
	`L`	Load vector
	`M`	Constraint vector
	`N`	Constraint Jacobian
	`D`	Damping matrix
	`E`	Mass matrix
	`NF`	Constraint force Jacobian
	`NP`	Optimization constraint Jacobian (*)
	`MP`	Optimization constraint vector (*)
	`MLB`	Lower bound constraint vector (*)
	`MUB`	Upper bound constraint vector (*)

Property/Value Pairs for the property out.

PROPERTY	EXPRESSION	DESCRIPTION
	Kc	Eliminated stiffness matrix
	Lc	Eliminated load vector
	Dc	Eliminated damping matrix
	Ec	Eliminated mass matrix
	Null	Constraint null-space basis
	Nullf	Constraint force null-space matrix
	ud	Particular solution ud
	uscale	Scale vector

(*) Requires the Optimization Module.

Note that the assembly of the eliminated matrices uses the current solution vector as scaling method. To get the unscaled eliminated system matrices, it is required to set the scaling method to `'none'` in the variables step of the solver configuration node.

The load vector is assembled using the current solution available as linearization point unless the `initmethod` property is provided. In case of the presence of a solver step node in the solver sequence, the load vector correspond then to the residual of the problem.

Example

Load `heat_convection_2d.mph` from the Model Library:

```
model = mphload('heat_convection_2d.mph');
```

Extract the stiffness matrix and the load vector:

```
str =
mphmatrix(model,'sol1','out',{'K','L'},'initmethod','init');
```

Plot the sparsity of the matrix:

```
spy(str.K)
```

Extract the eliminated system:

```
str =
mphmatrix(model,'sol1','out',{'Kc','Lc'},'initmethod','init');
```

Compare the sparsity of both system matrices (non-eliminated (b) and eliminated one (r)):

```
hold on
spy(str.Kc,'r')
```

Load heat_radiation_1d.mph from the model library:

```
model = mphload('heat_radiation_1d.mph');
```

Extract the eliminated residual:

```
str = mphmatrix(model,'sol1','out',{'Lc'});
```

See also mphstate, mphxmeshinfo

Purpose

Perform maximum of expressions.

Syntax

```
[v1,...,vn] = mphmax(model,{e1,...,en},edim,...)
[v1,...,vn,unit] = mphmax(model,{e1,...,en},edim,...)
```

Description

`[v1,...,vn] = mphmax(model,{e1,...,en},edim,...)` evaluates the maximum of the string expressions `e1,...,en` and returns the result in N matrices `v1,...,vn` with M rows and P columns. M is the number of inner solution and P the number of outer solution used for the evaluation. `edim` defines the element dimension: `line`, `surface`, `volume` or as an integer value.

`[v1,...,vn] = mphmax(model,{e1,...,en},edim,...)` also returns the units of the maximum in a 1xN cell array.

The function `mphmax` accepts the following property/value pairs:

TABLE 6-13: PROPERTY/VALUE PAIRS FOR THE MPHMAX COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Dataset`	`String`	`active solution data set`	Data set tag
`Matrix`	`off | on`	`on`	Returns data as a matrix or as a cell
`Outersolnum`	`Positive integer array`	`1`	Solution number for parametric sweep
`Selection`	`Integer vector | string | all`	`all`	Selection list or named selection
`Solnum`	`Integer vector | end | all`	`all`	Solution for evaluation
`Squeeze`	`on | off`	`on`	Squeeze singleton dimensions
`T`	`Double array`		Time for evaluation

The property `Dataset` controls which data set is used for the evaluation. Data Sets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution Data Sets.

The maximum expressions `e1,...,en` is evaluated for one or several solutions. Each solution generates an additional column in the returned matrix. The property `Solnum` and `t` control which solutions are used for the evaluation. The `Solnum` property is available when the data set has multiple solutions, for example in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is

available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

`Solnum` is used to select the solution number when a parametric, eigenvalue or time-dependent solver has been used.

`Outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

If the `Matrix` property is set to `off` the output will be cell arrays of length P containing cell arrays of length M.

Example

Load `micromixer.mph` from the Model Library:

```
model = mphload('micromixer.mph');
```

Find the maximum *x*-velocity `u` at the outlet boundary and get its unit:

```
[flow unit]= mphmax(model,'u','surface','selection',136)
```

Load `heat_transfer_axi.mph` from the Model Library:

```
model = mphload('heat_transient_axi.mph');
```

Find the max normal heat flux along the external boundaries:

```
Q = mphmax(model,'ht.ndflux','line','selection',[2,3,4]);
```

See also

`mphmean`, `mphmin`

Purpose

Perform mean of expressions.

Syntax

```
[v1,...,vn] = mphmean(model,{e1,...,en},edim,...)
[v1,...,vn,unit] = mphmean(model,{e1,...,en},edim,...)
```

Description

`[v1,...,vn] = mphmean(model,{e1,...,en},edim,...)` evaluates the means of the string expressions `e1,...,en` and returns the result in N matrices `v1,...,vn` with M rows and P columns. M is the number of inner solution and P the number of outer solution used for the evaluation. `edim` defines the element dimension: `line`, `surface`, `volume` or as an integer value.

`[v1,...,vn] = mphmean(model,{e1,...,en},edim,...)` also returns the units of the maximum in a 1xN cell array.

The function `mphmean` accepts the following property/value pairs:

TABLE 6-14: PROPERTY/VALUE PAIRS FOR THE MPHMEAN COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Dataset`	`String`	`active solution data set`	Data set tag
`Intorder`	`Positive integer`	`4`	Integration order
`Matrix`	`off | on`	`on`	Returns data as a matrix or as a cell
`Method`	`auto | integration | summation`	`auto`	Integration method
`Outersolnum`	`Positive integer array`	`1`	Solution number for parametric sweep
`Selection`	`Integer vector | string | all`	`all`	Selection list or named selection
`Solnum`	`Integer vector | end | all`	`all`	Solution for evaluation
`Squeeze`	`on | off`	`on`	Squeeze singleton dimensions
`T`	`Double array`		Time for evaluation

The property `Dataset` controls which data set is used for the evaluation. Data Sets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution Data Sets.

The mean of expressions `e1,...,en` is evaluated for one or several solutions. Each solution generates an additional column in the returned matrix. The property

`Solnum` and `t` control which solutions are used for the evaluation. The `Solnum` property is available when the data set has multiple solutions, for example in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

`Solnum` is used to select the solution number when a parametric, eigenvalue or time-dependent solver has been used.

`Outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

If the `Matrix` property is set to `off` the output will be cell arrays of length P containing cell arrays of length M.

Example

Load `micromixer.mph` from the Model Library:

```
model = mphload('micromixer.mph');
```

Find the mean *x*-velocity `u` at the outlet boundary and get its unit:

```
[flow unit]= mphmean(model,'u','surface','selection',136)
```

Load `heat_transfer_axi.mph` from the Model Library:

```
model = mphload('heat_transient_axi.mph');
```

Find the mean normal heat flux along the external boundaries:

```
Q = mphmean(model,'ht.ndflux','line','selection',[2,3,4]);
```

See also

`mphmax`, `mphmin`

Purpose

Plot a mesh in a MATLAB figure window.

Syntax

```
mphmesh(model)
mphmesh(model,meshtag,...)
```

Description

`mphmesh(model)` plots the mesh case in a MATLAB figure.

`mphmesh(model,meshtag,...)` plots the mesh case `meshtag` in a MATLAB figure.

The function `mphmesh` accepts the following property/value pairs:

TABLE 6-15: PROPERTY/VALUE PAIRS FOR THE MPHMESH COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Parent`	`Double`		Parent axis
`Edgecolor`	`Char`	`k`	Edge color
`Edgelabels`	`on | off`	`off`	Show edge labels
`Edgelabelscolor`	`Char`	`k`	Color for edge labels
`Edgemode`	`on | off`	`on`	Show edges
`Facealpha`	`Double`	`1`	Set transparency value
`Facelabels`	`on | off`	`off`	Show face labels
`Facelabelscolor`	`Char`	`k`	Color for face labels
`Facemode`	`on | off`	`on`	Show faces
`Meshcolor`	`Char`	`flat`	Color for face element
`Vertexlabels`	`on | off`	`off`	Show vertex labels
`Vertexlabelscolor`	`Char`	`k`	Color for vertex labels
`Vertexmode`	`on | off`	`off`	Show vertices

Example

Load the example model `shell_diffusion.mph` from the Model Library:

```
model=mphload('shell_diffusion.mph');
```

Plot the model mesh:

```
mphmesh(model,)
```

Plot the mesh with a colored element and transparency set to 0.5:

```
mphmesh(model, 'mesh1', ...
          'edgecolor', 'b', ...
          'facealpha',0.5,...
          'meshcolor','r');
```

See also

`mphgeom`, `mphmeshstats`, `mphplot`

Purpose

Return mesh statistics and mesh data information

Syntax

```
stats = mphmeshstats(model)
stats = mphmeshstats(model, meshtag)
[stats,data] = mphmeshstats(model, meshtag)
```

Description

`stats = mphmeshstats(model)` returns mesh statistics of the model mesh case in the structure `str`.

`stats = mphmeshstats(model, meshtag)` returns mesh statistics of a mesh case `meshtag` in the structure `str`.

`[stats,data] = mphmeshstats(model, meshtag)` returns in addition the mesh data information such as vertex coordinates and definitions of elements in the structure `data`.

The output structure `stats` contains the following fields:

TABLE 6-16: FIELDS IN THE STATS STRUCTURE

FIELD	DESCRIPTION
`Meshtag`	Mesh case tag
`Isactive`	Is the mesh node active
`Hasproblems`	Does the mesh have problems?
`Iscomplete`	Is the mesh built to completion?
`Sdim`	Space dimension
`Types`	Cell array with type names
`Numelem`	Vector with the number of elements for each type
`Minquality`	Minimum quality
`Meanquality`	Mean quality
`Qualitydistr`	Quality distribution (vector)
`Minvolume`	Volume/area of the smallest element
`Maxvolume`	Volume/area of the largest element
`Volume`	Volume/area of the mesh

The output structure `data` contains the following fields:

TABLE 6-17: FIELDS IN THE DATA STRUCTURE

FIELD	DESCRIPTOIN
`Vertex`	Coordinates of mesh vertices

TABLE 6-17: FIELDS IN THE DATA STRUCTURE

FIELD	DESCRIPTOIN
Elem	Cell array of definition of each element type
Elementity	Entity information for each element type

Example

Load busbar.mph from the Model Library

```
model = mphload('busbar');
```

Show the mesh distribution in a figure

```
stats = mphmeshstats(model);
bar(linspace(0,1,20), stats.qualitydistr)
```

Show the element vertices in a plot

```
[stats,data] = mphmeshstats(model);
plot3(data.vertex(1,:), data.vertex(2,:), ...
data.vertex(3,:), '.')
axis equal
view(3)
```

Get the element types information

```
stats.types
```

Get the number of edge element, note that the edge type is the first type in the list

```
numedgeelem = stats.numelem(1)
```

See also

mphmesh

Purpose

Perform minimum of expressions.

Syntax

```
[v1,...,vn] = mphmin(model,{e1,...,en},edim,...)
[v1,...,vn,unit] = mphmin(model,{e1,...,en},edim,...)
```

Description

`[v1,...,vn] = mphmin(model,{e1,...,en},edim,...)` evaluates the minimum of the string expressions `e1,...,en` and returns the result in N matrices `v1,...,vn` with M rows and P columns. M is the number of inner solution and P the number of outer solution used for the evaluation. `edim` defines the element dimension: `line`, `surface`, `volume` or as an integer value.

`[v1,...,vn] = mphmin(model,{e1,...,en},edim,...)` also returns the units in a 1xN cell array.

The function `mphmin` accepts the following property/value pairs:

TABLE 6-18: PROPERTY/VALUE PAIRS FOR THE MPHMIN COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`Dataset`	`String`	`active solution data set`	Data set tag
`Matrix`	`off | on`	`on`	Returns data as a matrix or as a cell
`Outersolnum`	`Positive integer array`	`1`	Solution number for parametric sweep
`Selection`	`Integer vector | string | all`	`all`	Selection list or named selection
`Solnum`	`Integer vector | end | all`	`all`	Solution for evaluation
`Squeeze`	`on | off`	`on`	Squeeze singleton dimensions
`T`	`Double array`		Time for evaluation

The property `Dataset` controls which data set is used for the evaluation. Data Sets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution Data Sets.

The mean of expressions `e1,...,en` is evaluated for one or several solutions. Each solution generates an additional column in the returned matrix. The property `Solnum` and `t` control which solutions are used for the evaluation. The `Solnum` property is available when the data set has multiple solutions, for example in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is

available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

`Solnum` is used to select the solution number when a parametric, eigenvalue or time-dependent solver has been used.

`Outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

If the `Matrix` property is set to `off` the output will be cell arrays of length P containing cell arrays of length M.

Example

Load `micromixer.mph` from the Model Library:

```
model = mphload('micromixer.mph');
```

Find the minimum *x*-velocity `u` at the outlet boundary and get its unit:

```
[flow unit]= mphmin(model,'u','surface','selection',136)
```

Load `heat_transfer_axi.mph` from the Model Library:

```
model = mphload('heat_transient_axi.mph');
```

Find the minimum normal heat flux along the external boundaries:

```
Q = mphmin(model,'ht.ndflux','line','selection',[2,3,4]);
```

See also

`mphmax`, `mphmean`

Purpose

Return tags for the nodes and subnodes in the COMSOL model object.

Syntax

```
mphmodel(model)
str = mphmodel(model,'-struct')
```

Description

`mphmodel(model)` returns the tags for the nodes and subnodes of the object `model`.

`str = mphmodel(model,'-struct')` returns the tags for the nodes and subnodes of the object `model` as a MATLAB structure `str`.

The function `mphmodel` can be used when navigating the model object and learning about its structure. The `mphmodel` function is mainly designed for usage when working on the command line and one needs to learn what nodes are placed under a particular node.

Example

Load `transport_and_adsorption.mph` from the Model Library:

```
model = mphload('transport_and_adsorption')
```

See what nodes are available under the model object:

```
mphmodel(model)
```

See what nodes are available under the geometry node:

```
mphmodel(model.geom)
```

Get the model information as a structure:

```
res = mphmodel(model, '-struct')
```

See also

`mphnavigator`, `mphsearch`

Purpose

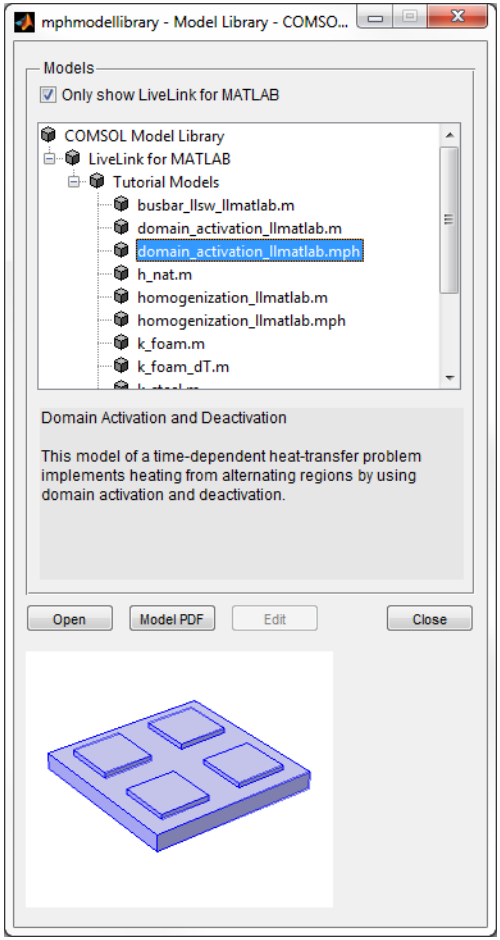
Graphical User Interface for viewing the Model Library.

Syntax

```
mphmodellibrary
```

Description

`mphmodellibrary` starts a GUI to visualize and access the example model available in the COMSOL Model Library. The model MPH-file can be loaded in MATLAB and the model documentation PDF-file is accessible directly. Models that are specific to LiveLink *for* MATLAB also contains the script M-file.



Purpose

Graphical User Interface for viewing the COMSOL model object

Syntax

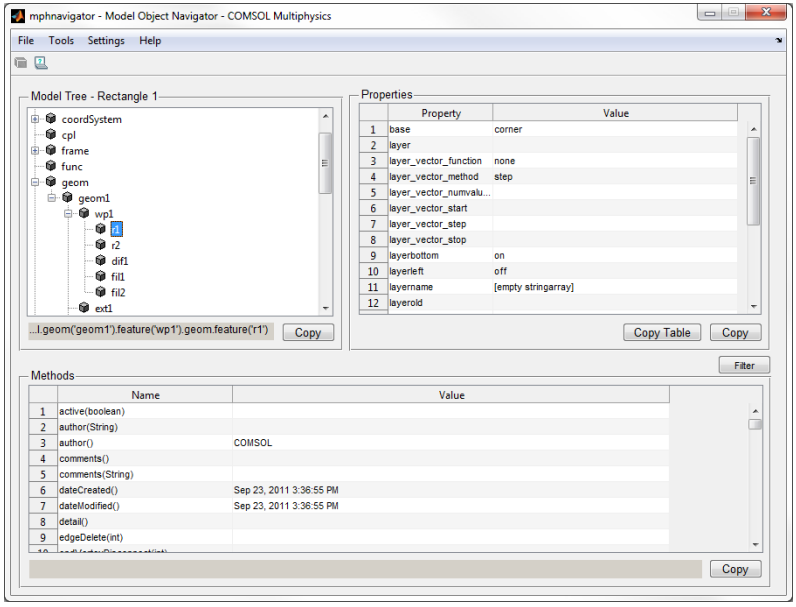
```
mphnavigator
mphnavigator(modelvariable)
```

Description

`mphnavigator` opens the Model Object Navigator which is a graphical user interface that can be used to navigate the model object and to view the properties and methods of the nodes in the model tree.

The GUI requires that the COMSOL objest is stored in a variable in the base workspace (at the MATLAB command prompt) with the name `model.`

`mphnavigator(modelvariable)` opens the model object defined with the name `modelvariable` in Model Object Navigator.



Example

Load `busbar.mph` from the Model Library:

```
model = mphload('busbar')
```

Navigate the model object that is accessible with the variable `model`

```
mphnavigator
```

Load `effective_diffusivity.mph` from the Model Library and set the model object with the variable `eff_diff`:

```
eff_diff = mphload('effective_diffusivity');
```

Navigate the model object that is accessible with the variable `eff_diff`

```
mphnavigator(eff_diff)
```

See also `mphgetexpressions`, `mphgetproperties`, `mphgetselection`, `mphmodel`, `mphsearch`

Purpose

Render a plot group in a figure window.

Syntax

```
mphplot(model,pgtag,...)
pd = mphplot(model,pgtag,...)
mphplot(pd,...)
```

Description

`mphplot(model,pgtag,...)` renders the plot group tagged `pgtag` from the model object `model` in a figure window in MATLAB.

`pd = mphplot(model,pgtag,...)` also returns the plot data used in the MATLAB figure in a cell array `pd`.

`mphplot(pd,...)` makes a plot using the post data structure `pd` that is generated using the function `mpheval`. Plots involving points, lines and surfaces are supported.

The function `mphplot` accepts the following property/value pairs:

TABLE 6-19: PROPERTY/VALUE PAIRS FOR THE MPHPLOT COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Colortable`	`String`	`Rainbow`	Color table used for plotting post data structure
`Index`	`Positive integer`	`1`	Index of variable to use plotting post data structure
`Rangenum`	`Positive Integer`	`none`	Color range bar (or legend) to display
`Server`	`on | off`	`off`	Plot on server
`Parent`	`Double`		Set the parent axes

Note: The plot on server option requires that you start COMSOL with MATLAB in graphics mode.

Only one color range bar and one legend bar is supported in a MATLAB figure. When the option plot on server is active, all active color range bar are displayed.

Example

Load `feeder_clamp.mph` from the Model Library:

```
model = mphload('feeder_clamp.mph');
```

Plot the first plot group

```
mphplot(model,'pg1')
```

Plot the first plot group with the color range bar:

```
mphplot(model,'pg1','rangenum',1)
```

Load fluid_valve.mph and plot on server (requires that you start COMSOL with MATLAB in graphics mode):

```
model = mphload('fluid_valve.mph');
```

Plot the second plot group on server:

```
mphplot(model,'pg2','server','on')
```

Load busbar.mph from the Model Library:

```
model = mphload('busbar.mph')
```

Extract temperature and electric potential data in the busbar domain:

```
pd = mpheval(model,{'T','V'},'selection',1);
```

Plot the electric potential data using the disco color table

```
mphplot(pd,'index',2,'colortable','disco')
```

See also colortable, mpheval

Purpose Save a COMSOL model

Syntax

```
mphsave(model)
mphsave(model, filename)
```

Description

`mphsave(model)` saves the COMSOL model object model.

`mphsave(model, filename)` saves the COMSOL model object `model` to the file named `filename`.

If the file name is not provided, the model has to be saved previously on disk.

If the file name does not provide a path, the file is saved relatively to the current path in MATLAB.

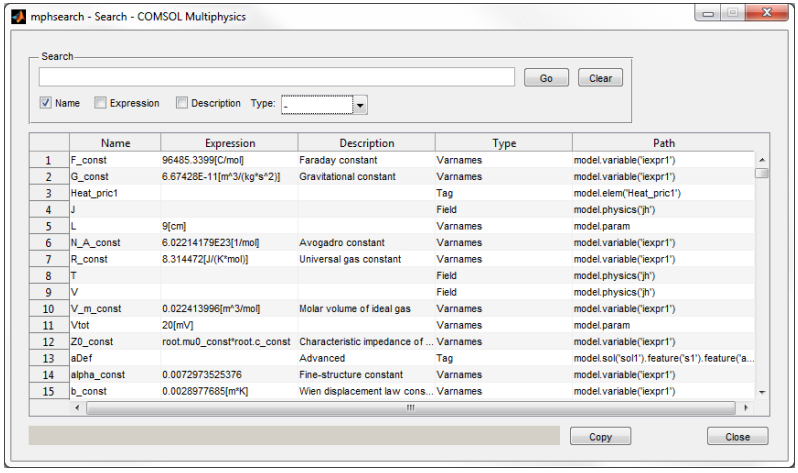
The model can be saved as an mph-file, java-file or as an m-file. The file extension determines which format that is saved.

See also `mphload`

Purpose GUI for searching expressions in the COMSOL model object

Syntax `mphsearch(model)`

Description `mphsearch(model)` opens a graphical user interface that can be used to search expressions in the model object `model`. Search using a text available in the name, expression or description of the variable.



	Name	Expression	Description	Type	Path
1	F_const	96485.3399[C/mol]	Faraday constant	Varnames	model.variable('iexpr1')
2	G_const	6.67428E-11[m^3/(kg*s^2)]	Gravitational constant	Varnames	model.variable('iexpr1')
3	Heat_pric1			Tag	model.elem('Heat_pric1')
4	J			Field	model.physics('jh')
5	L	9[cm]		Varnames	model.param
6	N_A_const	6.02214179E23[1/mol]	Avogadro constant	Varnames	model.variable('iexpr1')
7	R_const	8.314472[J/(K*mol)]	Universal gas constant	Varnames	model.variable('iexpr1')
8	T			Field	model.physics('jh')
9	V			Field	model.physics('jh')
10	V_m_const	0.022413996[m^3/mol]	Molar volume of ideal gas	Varnames	model.variable('iexpr1')
11	Vtot	20[mV]		Varnames	model.param
12	Z0_const	root.mu0_const*root.c_const	Characteristic impedance of ...	Varnames	model.variable('iexpr1')
13	aDef		Advanced	Tag	model.sol('sol1').feature('s1').feature('a...
14	alpha_const	0.0072973525376	Fine-structure constant	Varnames	model.variable('iexpr1')
15	b_const	0.0028977685[m*K]	Wien displacement law cons...	Varnames	model.variable('iexpr1')

See also `mphgetexpressions`, `mphnavigator`

Purpose

Select geometric entity using a rubberband/box.

Syntax

```
n = mphselectbox(model,geomtag,boxcoord,entity,...)
```

Description

`n = mphselectbox(model,geomtag,boxcoord,entity,...)` returns the indices of the geometry entities that are inside a selection domain (rectangle or box). This method looks only on the vertex coordinates and does not observe all points on curves and surfaces.

`boxcoord` set the coordinates of the selection domain, specified as a Nx2 array, where N is the geometry space dimension.

`entity` can be one of `'point'`, `'edge'`, `'boundary'` or `'domain'` following the entity space dimension defined below:

- `'domain'`: maximum geometry space dimension
- `'boundary'`: maximum geometry space dimension -1
- `'edges'`: 1(only for 3D geometry)

The function `mphpselectbox` accepts the following property/value pairs:

TABLE 6-20: PROPERTY/VALUE PAIRS FOR THE MPHSELECTBOX COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
Adjnumber	Scalar	none	Adjacent entity number

When a model uses *form an assembly* more than one vertex may have the same coordinate if the coordinate is shared by separate geometric objects. In that case one can use the adjnumber property in order to identify the domain that the vertices should be adjacent to.

Example

Load `busbar.mph` from the Model Library:

```
model = mphload('busbar.mph')
```

Find domains that are inside the selection box defined by [0,0.05;0,-0.05;-0.05,0.05]:

```
n = mphselectbox(model,'geom1',...
[0 0.05;0 -0.05;-0.05 0.05],'domain');
```

Find boundaries inside the selection box that are adjacent to domain number 1:

```
n = mphselectbox(model,'geom1',...
[0,0.05;0,-0.05;-0.05,0.05],'boundary','adjnumber',1);
```

Load `effective_diffusivity.mph` from the Model Library:

```
model = mphload('effective_diffusivity.mph');
```

Find the boundaries that are inside the selection rectangle defined by [3e-4,4e-4;4.5e-4,5.5e-4]:

```
n = mphselectbox(model,'geom1',...
[3e-4,4e-4;4.5e-4,5.5e-4],'boundary');
```

Find the boundaries that are inside the same selection rectangle:

```
n = mphselectbox(model,'geom1',...
[3e-4,4e-4;4.5e-4,5.5e-4],'point');
```

See also `mphgetadj`, `mphgetcoords`, `mphselectcoords`

Purpose

Select geometric entity using point coordinates

Syntax

```
n = mphselectcoords(model,geomtag,coord,entity,...)
```

Description

`n = mphselectcoords(model,geomtag,coord,entity,...)` finds geometry entity numbers based on their vertex coordinates.

One or more coordinate may be provided. The function searches for vertices near these coordinates using a tolerance radius. The list of the entities that are adjacent to such vertices is returned.

`Coord` is a NxM array where N correspond of the number of point to use and M the space dimension of the geometry.

`Entity` can be one of `'point'`, `'edge'`, `'boundary'` or `'domain'` following the entity space dimension defined below:

- `'domain'`: maximum geometry space dimension
- `'boundary'`: maximum geometry space dimension -1
- `'edges'`: 1(only for 3D geometry)

The function `mphpselectcoords` accepts the following property/value pairs:

TABLE 6-21: PROPERTY/VALUE PAIRS FOR THE MPHSELECTCOORDS COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Adjnumber`	`Scalar`	`none`	Adjacent entity number
`Radius`	`Scalar`	`auto`	Search radius
`Include`	`all | any`	`all`	Include all or any vertices

When a model uses *form an assembly* more than one vertex may have the same coordinate if the coordinate is shared by separate geometric objects. In that case one can use the adjnumber property in order to identify the domain that the vertices should be adjacent to.

The `radius` property is used to specify the radius of the sphere/circle that the search should be within. A small positive radius (based on the geometry size) is used by default in order to compensate for rounding errors.

If the `include` property is `'all'` then all vertices must belong to the entity in to be considered a match. If the `Include` property is `'any'` then an entity is considered a match as long as any of the vertices are adjacent to the entity.

Examplee

Load `busbar.mph` from the Model Library:

```
model = mphload('busbar.mph');
```

Select the vertex near [0.095,0,0.1]:

```
n = mphselectcoords(model,'geom1',... [0.095,0,0.1],'point');
```

Select the edge from [0.095,0,0.1] to [0.095,0,0.01]:

```
n = mphselectcoords(model, 'geom1',...
[0.095,0,0.1;0.095,0,0.01]','edge');
```

Select edges that are adjacent to the points [0.095,0,0.1] and [0.095,0,0.01]:

```
n = mphselectcoords(model, 'geom1',...
[0.095,0,0.1;0.095,0,0.01]','edge',... 'include', 'any');
```

Select boundaries that are adjacent to the points [0.09,0,0.1] and [0.09,0,0.015] with a search radius of 0.01:

```
n = mphselectcoords(model,'geom1',...
[0.09,0,0.1;0.09,0,0]','boundary',... 'radius',0.01);
```

See also mphgetadj, mphgetcoords, mphselectbox

Purpose Show messages in error nodes in the COMSOL model

Syntax

```
mphshowerrors(model)
list = mphshowerrors(model)
```

Description `mphshowerrors(model)` shows the error and warning messages stored in the model and where they are located. The output is displayed in the command window.

`list = mphshowerrors(model)` returns the error and warning messages stored in the model and where they are located in the Nx2 cell array list. N corresponding to the number of errors or warning found in the model object. The first column contains the node of the error and the second column contain the error message.

See also `mphnavigator`

Purpose

Get information about a solution object

Syntax

```
info = mphsolinfo(model,...)
info = mphsolinfo(model,'solname',soltag,...)
```

Description

`info = mphsolinfo(model,...)` returns information about the default solution object.

`info = mphsolinfo(model,'solname',soltag,...)` returns information about the solution object `soltag`.

The function `mphsolinfo` accepts the following property/value pairs:

TABLE 6-22: PROPERTY VALUE PAIRS FOR THE MPHSOLINFO COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Solname`	`String`	Active solution object	Solution object tag
`Dataset`	`String`	`Active solution data set`	Data set tag
`NU`	`on | off`	`off`	Get info about number of solutions

The returned value `info` is a structure with the following content

TABLE 6-23: FIELDS IN THE INFO STRUCT

FIELD	CONTENT
`Solname`	Solution name
`Size`	Size of the solution vector
`Nummesh`	Number of meshes in the solution (for automatic remeshing)
`Sizes`	Size of the solution vector for each mesh and number of timesteps/parameters for each mesh
`Soltype`	Solver type (Stationary, Parametric, Time or Eigenvalue)
`Solpar`	Name of the parameter
`Sizesolvals`	Length of the parameter list
`Solvals`	Values of the parameters, eigenvalues or timesteps
`Paramsweepnames`	Parametric sweep parameter names
`Paramsweepvals`	Parametric sweep parameter values
`NUsol`	Number of solution vectors stored
`NUreacf`	Number of reaction forces vectors stored

TABLE 6-23: FIELDS IN THE INFO STRUCT

FIELD	CONTENT
NUadj	Number of adjacency vectors stored
NUfsens	Number of functional sensitivity vectors stored
NUsens	Number of forward sensitivity vectors stored

You can use the function `mphgetu` to obtain the actual values of the solution vector. Note that these functions are low level functions and you most often would use functions such as `mphinterp` and `mpheval` to extract numerical data from a model.

Examplee

Load `stress_in_pulley.mph` from the Model Library

```
model = mphload('stress_in_pulley.mph');
```

Get the information of the active solution object

```
info = mphsolinfo(model);
```

Get the information of the second solution object

```
info = mphsolinfo(model,'solname',sol2);
```

See also `mphgetu`, `mphxmeshinfo`

Purpose

Connect MATLAB to a COMSOL server.

Syntax

```
mphstart
mphstart(port)
mphstart(ipaddress, port)
mphstart(ipaddress, port, comsolpath)
```

Description

`mphstart` creates a connection with a COMSOL server using the default port number (which is 2036).

`mphstart(port)` creates a connection with a COMSOL server using the specified port number `port`.

`mphstart(ipaddress, port)` creates a connection with a COMSOL server using the specified IP address `ipaddress` and the port number `port`.

`mphstart(ipaddress, port, comsolpath)` creates a connection with a COMSOL server using the specified IP address and port number using the `comsolpath` that is specified. This is useful if `mphstart` cannot find the location of the COMSOL Multiphysics installation.

`mphstart` can be used to create a connection from within MATLAB when this is started without using the *COMSOL with MATLAB* option. `mphstart` then sets up the necessary environment and connect to COMSOL.

Prior to calling `mphstart` it is necessary to set the path of `mphstart.m` in the MATLAB path or to change the current directory in MATLAB (for example, using the cd command) to the location of the mphstart.m file.

A COMSOL server must be started prior to running `mphstart`.

Purpose

Get state-space matrices for dynamic system.

Syntax

```
str = mphstate(model,soltag,'Out',{'SP'})
  str = mphstate(model,soltag,'Out',{'SP1','SP2',...})
```

Description

`str = mphstate(model,soltag,'out',{'SP'})` returns a MATLAB structure `str` containing the state space matrix `SP` assembled using the solver node `soltag` and accessible as `str.SP`. `SP` being taken from the `Out` property list.

`str = mphstate(model,soltag,'Out',{'SP1','SP2',...})` returns a MATLAB structure `str` containing the state space matrices `SP1`, `SP2`,... assembled using the solver node `soltag` and accessible as `str.SP1`and `str.SP2`. `SP1` and `SP2` being taken from the `Out` property list.

The function `mphstate` accepts the following property/value pairs:

TABLE 6-24: PROPERTY VALUE FOR THE MPHSTATE COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Out`	`MA | MB | A | B | C | D |Mc |Null | ud | x0`		Output matrix
`Keepfeature`	`off | on`	`off`	Keep the state-space feature in the model
`Input`	`String`		Input variables
`Output`	`String`		Output variables
`Sparse`	`off | on`	`off`	Return sparse matrices
`Initmethod`	`init | sol`		Use linearization point
`Initsol`	`solname | zero`	`solname`	Solution to use for linearization
`Solmum`	`Positive integer`	`auto`	Solution number

The property `Sparse` controls whether the matrices `A`, `B`, `C`, `D`, `M`, `MA`, `MB`, and `Null` are stored in the sparse format.

The equations correspond to the system below:

$$\begin{cases} Mc\dot{x} = McAx + McBu \\ y = Cx + Du \end{cases}$$

where x are the state variables, u are the input variables, and y are the output variables.

A static linearized model of the system can be described by:

$$y = (D - C(McA)^{-1}McB)u$$

The full solution vector U can be then obtained from

$$U = \text{Null}x + ud + u0$$

where Null is the null space matrix, ud the constraint contribution and $u0$ is the linearization point, which is the solution stored in the sequence once the state space export feature is run.

The matrices Mc and MA are produced by the same algorithms that do the finite-element assembly and constraint elimination in COMSOL Multiphysics. Mc and MA are the same as the matrices D_c (eliminated mass matrix) and $-K_c$ (K_c is the eliminated stiffness matrix). The matrices are produced from an exact residual vector Jacobian calculation (that is, differentiation of the residual vector with respect to the degrees of freedoms x) plus an algebraic elimination of the constraints. The matrix C is produced in a similar way; that is, the exact output vector Jacobian matrix plus constraint elimination.

The matrices MB and D are produced by a numerical differentiation of the residual and output vectors, respectively, with respect to the input parameters (the algorithm systematically perturbs the input parameters by multiplying them by a factor $(1+10^{-8})$).

The input cannot be a variable constraint in the model.

Example

Load `heat_transient_axi.mph` from the Model Library

```
model = mphload('heat_transient_axi.mph');
```

Set the temperature condition using a parameter

```
model.param.set('Tinput','1000[degC]');
temp1 = model.physics('ht').feature('temp1');
temp1.set('T0',1,'Tinput');
```

Add a domain point probe at (0.28; 0.38)

```
pdom1 = model.probe.create('pdom1,'DomainPoint');
pdom1.model('mod1');
pdom1.setIndex('coords2','0.28',0,0);
pdom1.setIndex('coords2','0.38',0,1);
```

Extract the state-space matrix:

```
str = mphstate(model,'sol1','out',...
{'MA','MB','C','D'},'input','TO',... 'output','mod1.ppb1');
```

Purpose Return the version number of COMSOL Multiphysics

Syntax

```
v = mphversion
[v,vm] = mphversion(model)
```

Description

`v = mphversion` returns the COMSOL Multiphysics version number that MATLAB is connected to as a string.

`[v,vm] = mphversion(model)` returns the COMSOL Multiphysics version number that MATLAB is connected to as a string in the variable `v` and the version number of the model in the variable `vm`.

See also `mphload`, `mphsave`

Purpose Display a geometric entity selection in a MATLAB figure.

Syntax

```
mphviewselection(model,geomtag,number,'entity', entity,...)
mphviewselection(model,seltag,...)
```

Description `mphviewselection(model,geomtag,number,'entity', entity,...)` displays the geometric entity `number` of type `entity` in MATLAB figure including the representation of the geometry `geomtag`.

`mphviewselection(model,seltag,...)` displays the geometric entity selection `seltag` in a MATLAB figure including the representation of the geometry.

The function `mphviewselection` accepts the following property/value pairs:

TABLE 6-25: PROPERTY VALUE/PAIRS FOR THE MPHVIEWSELECTION FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Edgecolor`	`Char | RGB array`	`k`	Color for edges
`Edgecolorselected`	`RGB array`	`[1,0,0]`	Color for selected edges
`Edgelabels`	`on | off`	`off`	Show edge labels
`Edgelabelscolor`	`Char | RGB array`	`g`	Color for edge labels
`Edgemode`	`on | off`	`on`	Show edges
`Entity`	`Domain | boundary | edge | point`		Set the selected entity type
`Facealpha`	`Double`	`1`	Set transparency value
`Facecolor`	`RGB array`	`[0.6,0.6,0.6]`	Color for face
`Facecolorselected`	`RGB array`	`[1,0,0]`	Color for selected face
`Facelabels`	`on | off`	`off`	Show face labels
`Facelabelscolor`	`Char | RGB array`	`b`	Color for face labels
`Facemode`	`on | off`	`on`	Show faces
`Geommode`	`on | off`	`on`	Show entire geometry
`Marker`		`.`	Vertex marker
`Markercolor`	`Char | RGB array`	`b`	Color for vertex marker
`Markercolorselected`	`Char | RGB array`	`r`	Color for selected vertex marker

TABLE 6-25: PROPERTY VALUE/PAIRS FOR THE MPHVIEWSELECTION FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
Markersize	Int	12	Font size of marker
Parent	Double		Parent axis
Renderer	Opengl | zbuffer	opengl	Set the rendering method
Selection	String | Positive integer array		Set selection name or entity number
Selectoralpha	Double	0.25	Set selector transparency value
Selectorcolor	RGB array	[0,0,1]	Color for selected marker
Showselector	on | off	on	Show Selector
Vertexlabels	on | off	off	Show vertex labels
Vertexlabelscolor	Char | RGB array	r	Color for vertex labels
Vertexmode	on | off	off	Show vertices

Example

Load busbar.mph from the Model Library:

```
model = mphload('busbar')
```

Plot boundary number 3 using a yellow color

```
mphviewselection(model,'geom1',3,'entity',...
'boundary','facecolorselected',[1 1 0],... 'facealpha', 0.5)
```

Plot edge 1 to 9 in green color

```
hold on
mphviewselection(model,'geom1',1:8,...
'geommode','off','entity','edge',... 'edgecolorselected', [0 1
0.5])
```

Plot the titanium blot domains in green color

```
mphviewselection(model,'sel1',... 'facecolorselected',[0 1 0])
```

See also

mphgeom, mphselectbox, mphselectcoords

Purpose

Extract information about the extended mesh.

Syntax

```
info = mphxmeshinfo(model)
```

Description

The Xmesh information provide information about the numbering of elements, nodes, and degrees of freedom (DOFs) in the extended mesh and in the matrices returned by `mphmatrix` and `mphgetu`

Information is only available on StudyStep and Variables features.

The function `mphxmeshinfo` accepts the following property/value pairs:

TABLE 6-26: PROPERTY VALUE/PAIRS FOR THE MPHVIEWSELECTION FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Solname`	`String`	`Active solution object`	Solution object tag
`Studysteptag`	`String`		Study step node tag
`Meshcase`	`Positive integer | String`	`First mesh`	Mesh case tag

The function `xmeshinfo` returns a structure with the fields shown in the table below

TABLE 6-27: FIELD IN THE RETURNED STRUCTURE FROM MPHXMESHINFO

FIELD	DESCRIPTION
`Solname`	Tag of the solution object
`Ndofs`	Number of DOFs
`Fieldnames`	Names of the field variables
`Fieldndofs`	Number of DOFs per field name
`Meshtypes`	Types of mesh element
`Dofs`	Structure with information about the degrees of freedom
`Nodes`	Struct with information about the nodes
`Elements`	Struct with information about each element type

Example

Load `busbar.mph` from the Model Library:

```
model = mphload('busbar.mph')
```

Extract xmesh information for the active Solver feature:

```
info = mphxmeshinfo(model)
```

Extract xmesh information from the Study Step node `st1`

```
info = mphxmeshinfo(model,'solname','sol1','studysteptag','st1')
```

See also: mphgetu, mphmatrix, mphsolinfo

Index

A access methods 243
adaptive solver 227
advancing front method 57
animations 110
ASCII file 111
ASCII format 110

B Boolean operations 39
boundary layer meshes 66

C CAD formats 42
circle 36
Compose operation 36
COMSOL API 26
COMSOL Desktop 26
exchanging geometries with 41
COMSOL Multiphysics binary files 73
COMSOL Multiphysics text files 73
constructor name 81
converting meshes 72
copying boundary meshes 70

D data sets
syntax for 108
Delaunay method 57
Difference operation 36
documentation, finding 12
DXF files 42

E emailing COMSOL 13
equations
global 86
modifying 85
evaluating data 109
exporting data for file 110
extruding, meshes by sweeping 64

F file formats
.mphbin files 42
.mphtxt files 42
fluid flow 66
free quad mesh 58
function inputs/outputs 183

G GDS format 42
geometries
1D 34
2D 35
3D 39
exchanging with the COMSOL Desktop 41
parameterized 44
geometry sequences 32
`getType()` methods 243

I importing meshes 73
Internet resources 11

J Java 26

K knowledge base, COMSOL 14

M mass matrix 150
materials 84
mesh
advancing front 57
converting 72
copying 70
creating a quad mesh 58
creating boundary layers 66
Delaunay 57
getting information about 76
importing 73
refining 69
mesh resolution 54
meshing sequences 52
Model Builder 26
Model Library 12

model object 26
ModelUtil 27
modifying equations 85
MPH-files 13

N NASTRAN files 73
NASTRAN mesh 73
native file formats 42
node point coordinates 119
no-slip boundary condition 66

P parameterized geometries 44
physics interfaces 81
plot groups
  syntax for 104

R rectangle 36
refining meshes 69
reports 110
results analysis 104
results evaluation 109

S sequences of operations 26
set operations 36
simplex elements 69
solver configurations
  syntax for 99
sparsity, of matrix 228
squeezed singleton 126
state-space matrices
  example of extracting 153
structured meshes 59
study, syntax for 97
summary of commands 188
sweeping, to revolve meshes 63
syntax
  for materials 84
  for physics interfaces 81

T technical support, COMSOL 13
tolerance radius 92
trimming solids 37
typographical conventions 14

U user community, COMSOL 14

V visualization 104

W web sites, COMSOL 14
weights, of control polygon 38